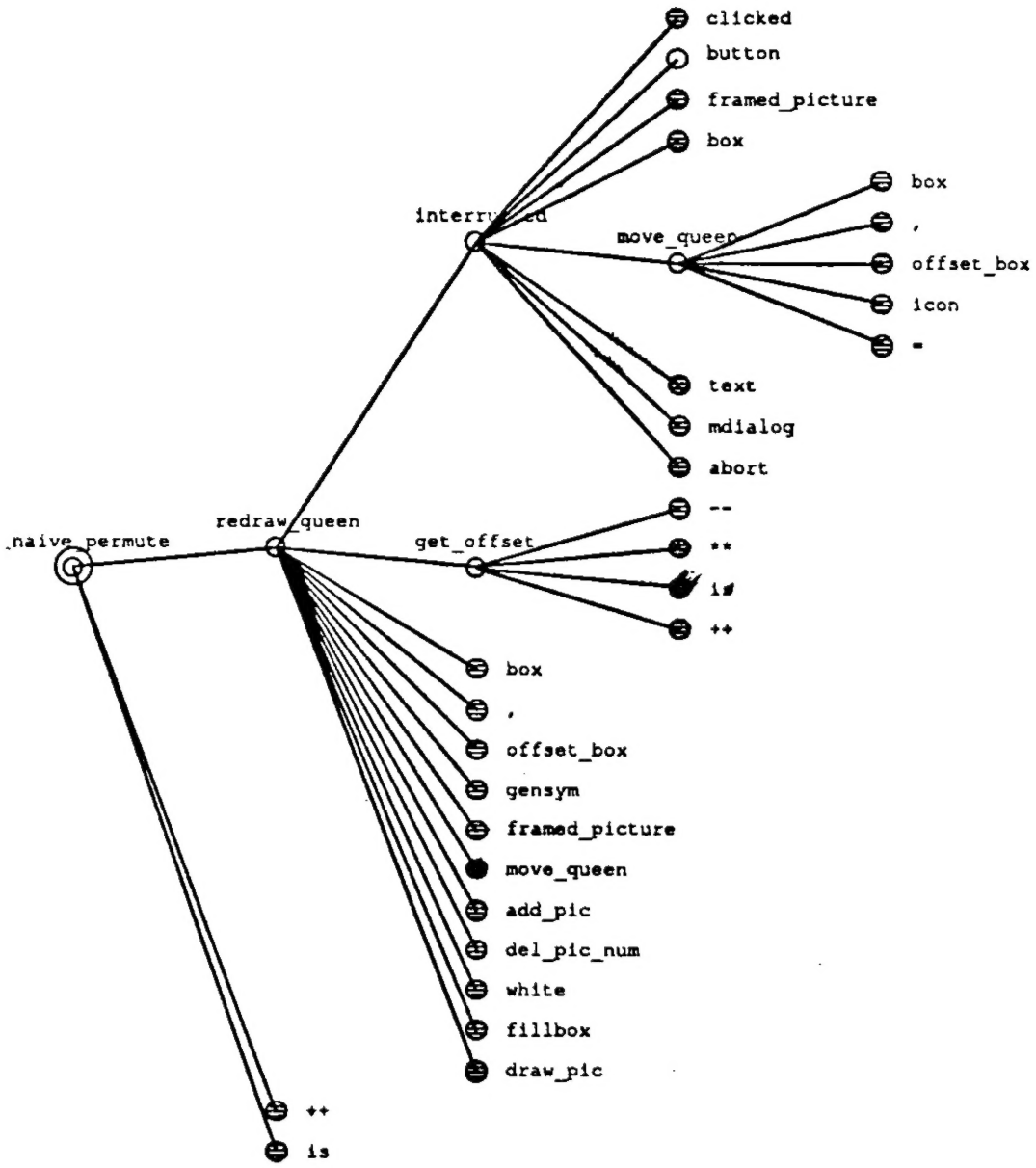
LPA MacPROLOG™ 2.0

Environment Guide




```
naive_permute([], [], Row, Speed, Win).
naive_permute(Cols, [Col|Final], Row, Speed, Win) :-
  remove(Col, Cols, Rest),
  redraw_queen(Speed, Win, Row, Col),
  Nrow is Row++1,
  naive_permute(Rest, Final, Nrow, Speed, Win).
```

LPA MacPROLOG™ Environment Guide

Nicky Johns
Phil Vasey

Logic Programming Associates Ltd
Studio 4
The Royal Victoria Patriotic Building
Trinity Road
London SW18 3SH

Read on ...

LPA MacPROLOG™ Environment Guide

Contents

Part A : Introduction to the Environment

Part B : Menu Reference

LPA MacPROLOG™

Environment Guide

If the Macintosh is new to you, you should first consult the Macintosh owner's guide. This covers in more detail those aspects of the user environment (such as Menus, Windows and how to use the keyboard and mouse) which are common to all Macintosh applications.

Part A : Introduction to the Environment

1. About this guide

This is a two-part introductory guide to the LPA MacPROLOG™ programming environment. It is not intended to teach you how to program in PROLOG, although we will of course be using PROLOG programs to demonstrate the various features of the environment. For an introduction to PROLOG programming we suggest that you study the book

> **PROLOG Programming for Artificial Intelligence**
> **by Ivan Bratko**
> **published by Addison-Wesley**

or the book

> **Programming in PROLOG**
> **by Clocksin & Mellish**
> **published by Springer-Verlag**

together with the chapter on Edinburgh Syntax in the **LPA MacPROLOG™ Reference Manual.**

For a description of the extensive range of primitives (built-in routines), you should consult the LPA MacPROLOG™ Reference Manual. In addition to the standard PROLOG primitives, there is a wide range of primitives allowing you to use the mouse, menu, window and dialogue features of the Macintosh so you can build advanced Macintosh style applications written solely in MacPROLOG. If you have the Graphics package, you can also build sophisticated graphics applications.

The guide is in two parts. Part A provides an introduction to the programming environment, and Part B is a summary of the menus of the environment.

Part A

The first section of this part shows you how to start MacPROLOG, and gives a brief summary of the Macintosh terms that we shall be using here and in the Reference Manual.

The second section takes you through the writing of a simple program in MacPROLOG, with the intention of introducing you to many of the features of the MacPROLOG programming environment. After you have worked through this example you will be ready to start developing your own programs.

Following this we describe some further features of the environment, including error handling. We introduce a more advanced program example written in MacPROLOG, and look at the debugging and tracing facilities available.

There is then an introduction to Graphic Windows and Call Graphs, available if you have the MacPROLOG Graphics package.

A final section deals with printing, using the Optimising Compiler, and the porting of PROLOG programs to and from MacPROLOG.

Part B

Part B provides a summary of all the top level menus and a full description of their use.

2. About MacPROLOG

To use MacPROLOG Version 2 you need a MacPlus, a Mac SE or a Mac II, with at least 1M memory and preferably an external drive. The more memory you have the better. MacPROLOG automatically makes use of all the memory that is available. You may also use MacPROLOG with a hard disk for increased speed and efficiency.

If you intend to sell applications developed using LPA MacPROLOG™ you must obtain a special licence from LPA enabling you to distribute the MacPROLOG run-time system.

3. Installing LPA MacPROLOG™ on your system

3.1 What is on the disk

MacPROLOG will have been delivered to you on a double-sided (800K) disk. This master disk contains:

1. The core MacPROLOG system - in the file **LPA MacPROLOG™**.
2. The compiled MacPROLOG code that defines the programming environment. This is in the file **MacPROLOG™ Boot**. A file of this name must always be on the disk that contains the file **LPA MacPROLOG™**.
3. An **Examples Folder** that contains several example programs, including the **Guide Example** used later in this guide.
4. A **Utilities Folder** containing some utility programs.
5. Two text files named **Menu Help** and **Primitives Help** which are used by the on-line help procedures.
6. A file named **Optimizer Boot** (if you have purchased the optimising compiler).
7. A file named **Graphics Boot** (if you have purchased the graphics interface).

There may also be a MacWrite file named **Read Me** which contains supplementary documentation.

3.2 Copying the disk

You should take a copy of the **LPA MacPROLOG™** disk. This guide assumes that you have done this.

You can also copy all the files of the MacPROLOG system onto a hard disk to benefit from the improved performance and space.

NOTE : For your convenience MacPROLOG is not copy protected. However LPA trusts you to abide by our licence agreement.

3.3 Loading LPA MacPROLOG

When you run MacPROLOG you should always use the *copy* that you have made of the master disk.

After booting up your Mac with a system disk, insert your LPA MacPROLOG™ disk into one of the drives, and double click on the **LPA MacPROLOG™** icon.

LPA MacPROLOG™

Double clicking on **LPA MacPROLOG™** causes the compiled program in **LPA MacPROLOG™ Boot** to be automatically loaded. This will boot you into the programming environment. Alternatively, double clicking on any MacPROLOG file (whether source or object code) also automatically loads the necessary **LPA MacPROLOG™** application files.

WARNINGS:
The actual name of the environment file **MacPROLOG™ Boot** (and also **Optimizer Boot** and **Graphics Boot** where present) is crucial and should *not* be changed!

You will *never* need to load the **Boot** files explicitly. They are always automatically loaded by MacPROLOG as they are needed.

4. Glossary

This is a brief glossary of some of the Macintosh terms we shall be using in this documentation. If you are familiar with the Macintosh, you can probably skip this section.

button
A round-cornered rectangular labelled box displayed inside a *dialogue*. Clicking inside a button normally terminates the dialogue.

check box
A small square box with a label, displayed inside a *dialogue*. Clicking on it selects or deselects it. When selected the box has a cross in it.

cursor
The flashing vertical bar in an edit window. The 'cursor' may also be a *selection range*. (Sometimes we may also use 'cursor' to mean the arrow or other symbol associated with the mouse.)

dialogue
A temporary window which is used to display or accept information. It usually contains *buttons* which can be clicked on using the mouse. Pressing the *Return* key is equivalent to clicking the **Ok** button, and pressing the full stop key with the *Command* key held down is equivalent to clicking the **Cancel** button. Dialogues may be *modal* or *modeless* (see below).

edit field
A rectangular box displayed inside a *dialogue* into which you may type text.

Finder
The Macintosh program which controls the 'desktop', i.e. before you have loaded an application such as MacPROLOG.

front window
The currently 'active' window on the screen, at the front of the display. Its title bar will have black stripes across it.(If it is a dialogue its title bar will be black.)

goaway box
A box on the left of the title bar in a window. Clicking in this box hides the window.

modal dialogue
A dialogue which must be responded to before anything else is done. A beep is sounded if an attempt is made to click anywhere outside the dialogue window.

modeless dialogue
A dialogue which can be responded to after doing other things. It can be moved around the screen and does not need to be kept as the front window.

radio button
A small circle with a label, displayed inside a *dialogue*. Clicking on it selects or deselects it. When selected the circle is filled. Normally only one of any group of radio buttons can be selected but you can select more than one by holding the Shift key down while you click. You can also deselect a selected radio button by shift clicking.

resize box
A small box in the bottom right hand corner of a window. Clicking and dragging this box will resize the window.

scroll bar
A vertical or horizontal bar along the edge of a window. Clicking in the arrows at either end of the scroll bar will scroll the window's contents by a small amount. Dragging the 'elevator box' will scroll by a larger amount.

selection range
Text that is displayed in reverse video. You can select text by clicking and dragging the mouse.

zoom box
A small box on the right of a window's title bar. Clicking in this box expands the window to full screen size. Clicking again shrinks the window to its original size.

5. Programming in MacPROLOG

In this next section we take you through a short MacPROLOG sample program, demonstrating some of the features of the windows, menus and dialogues of the MacPROLOG programming environment. The example used develops an elementary data base application.

We strongly recommend that you work through this example. The amount of typing you have to do has been kept to a minimum, but you will see how just a few simple lines of PROLOG can access a whole host of exciting Macintosh features! You will also very quickly become familiar with the style of programming required in MacPROLOG's multi-window environment.

At this stage we assume that you have at least an elementary knowledge of PROLOG (this tutorial is not intended to teach you PROLOG), and that you know how to use the standard Macintosh keyboard, mouse, menus and windows.

Throughout this example the intention is to give you a glimpse of some of the enhanced programming possibilities of MacPROLOG. The environment is powerful and sophisticated and we cannot possibly explore every avenue or go into every minor detail. If you find you have made a mistake, an error message should appear and will probably be fairly self-explanatory. If in doubt, click on the nearest available **Cancel** or **Stop** button and start again!

The example shows you some of the power of MacPROLOG for developing Macintosh applications. But the best demonstrator of this is the programming environment itself - the whole environment, with the single exception of the **File** menu, is implemented in MacPROLOG!

5.1 The MacPROLOG Programming Environment

Having loaded MacPROLOG you are now in the programming environment. At the bottom of the screen is the **Default Output Window**. This is where the answers to your PROLOG queries and evaluations will normally appear. You can't type into this window (although you can scroll its text and do other editing functions such as **Cut** and **Copy** from it).

You will also see the names of various menus in the menu bar across the top of the screen. We will be using some of these in this example; the others you will find explained in Part B of this guide.

5.2 Getting Finished

When you want to finish, select the **Quit** option from the **File** menu. If you have made any changes to your program since the last time it was saved you will be asked if you want to save the program before you go. Saving programs is described later.

5.3 Entering a PROLOG Query

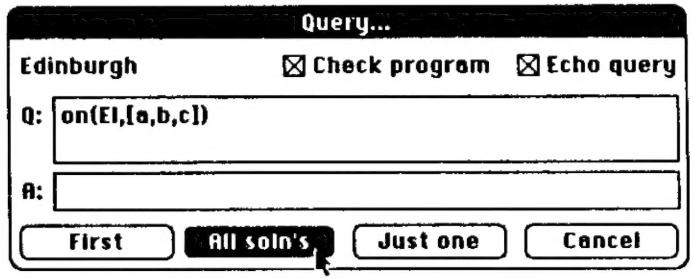
You can run a query by selecting **Query...** from the **Eval** menu. This displays the **Query...** dialogue. The **Q:** field is for you to enter your query, and the **A:** field is for specifying an answer pattern. At the start of a MacPROLOG session the **Q:** field contains **true**.

Type into the **Q:** field a query such as

```
on(El,[a,b,c])
```

and click the **All soln's** button.

(on, which is the list membership relation, is a primitive of MacPROLOG. `El` is a variable since it begins with an uppercase letter).

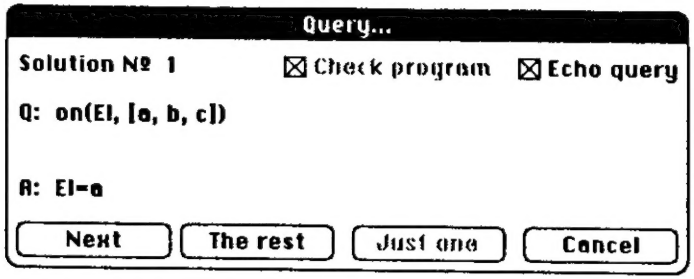


When you do this the **Query...** dialogue disappears, and all the solutions, together with the original query, appear in the **Default Output Window** (you may need to scroll or resize the window to see all the solutions).

Select **Query...** again and you will see that what you typed last time is still there. You can edit the query, type a different one, or just leave the **Q:** field exactly as it is. This time click on one of the **First** or **Just one** buttons.

You will find that the **Just one** option gives you 'just one' solution!

The **First** option displays the first solution in the **Default Output Window**, and the **Query...** dialogue changes to also display this solution, and offers **Next** and **The rest** buttons.



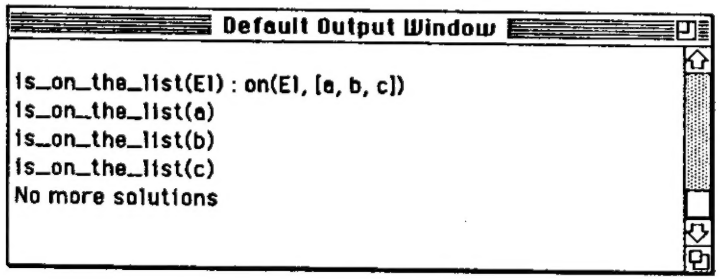
If you click on the **Next** button then the solutions are displayed one by one until there are no more solutions.

If you click on the **The rest** button then this is like choosing **All soln's** in the original dialogue: the **Query...** dialogue disappears and the rest of the solutions are displayed in the **Default Output Window**.

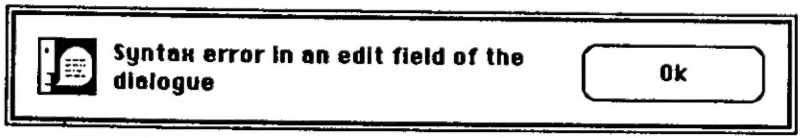
The **A:** field is for specifying an answer pattern. If in the above example you type

```
is_on_the_list(El)
```

in the **A:** field then the solutions will be displayed as



The **A:** field must contain a valid PROLOG term. If it does not then you will be told



You must then click the **Ok** button and correct what you have typed in the **A:** field.

If you leave the **A:** field blank then the bindings for *all* the variables in your query will be displayed as the answer.

You will notice that there are also two check boxes at the top of the **Query...** dialogue. They are both initially selected.

If you click on the **Echo query** box (to deselect it), then output is not sent to the **Default Output Window** but answers are displayed in the modified **Query...** dialogue described above. You can then step through solutions without them being echoed elsewhere.

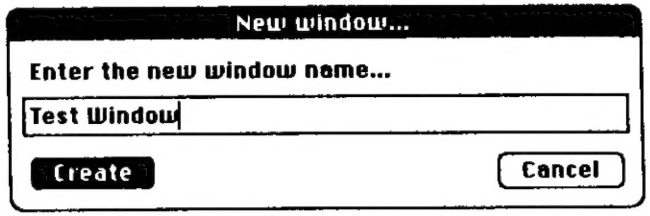
(If you choose the **All soln's** option with **Echo query** off this often means that you won't see the solutions to your query because they will flash by so quickly! It only makes sense to do this if each solution produces a different visible side effect, for example.)

The **Check program** box determines whether recompilation should take place before running a query. We will come back to this later.

5.4 Writing a PROLOG program

To write a program you type the source text into a program edit window, and then compile it.

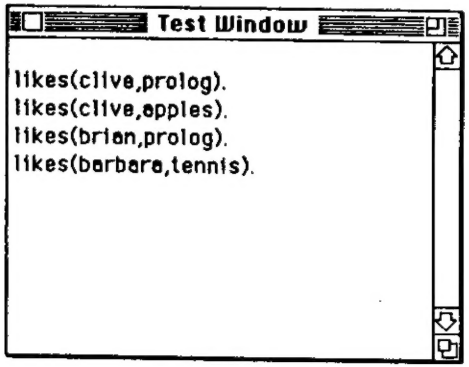
Select **New...** from the Windows menu. A dialogue will be displayed on the screen asking you for the name of the new window.



Type in a suitable name (we will use the name **Test Window** here) and click on the **Create** button.

Your new program window will appear on the screen, with the cursor in the top left hand corner. Anything you type will now appear in the window.

Type in the following facts.



To compile the window select **Check program** from the **Eval** menu. You will see a dialogue box at the bottom of the screen saying that the window is being compiled.

(If there are any errors in the window the offending term will be highlighted and there will be a dialogue displaying a Syntax Error message. Click on the **Stop** button, correct the error, and then do **Check program** again.)

Now enter the query

```
likes(clive,Something)
```

using the **Query...** command. You should get the solutions `prolog` and `apples` displayed.

Note: You can't call a MacPROLOG program by entering a query in an edit window. In MacPROLOG, other than using the **Query...** command, you can only call programs by mouse selection of a menu command or a graphic tool. These latter two methods are described later.

Recall that the **Query...** dialogue has a **Check program** check box. When this is selected, any necessary recompilation is done *automatically* before the query is evaluated. This means you don't have to explicitly do a **Check program** from the **Eval** menu before entering a query.

(Normally you will have several program windows. For increased speed and efficiency the **Check program** command only recompiles those windows which have been changed since the last recompilation.)

You can also enter a conjunction of terms in the **Query...** dialogue. Typing

```
likes(clive,Something),likes(brian,Something)
```

in the **Q:** field will produce the single solution `prolog`. If we extend the query further, to

```
likes(clive,Something),likes(brian,Something),
   likes(barbara,Something)
```

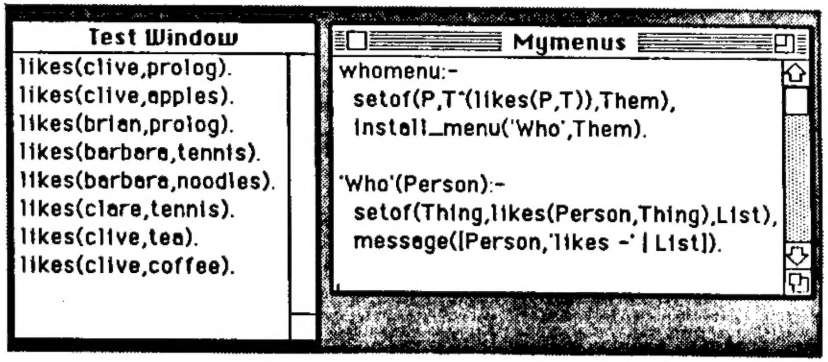
then we will get no solutions.

5.5 Menus and Dialogues

We will now extend this small data base and create a menu based on the facts we have entered.

First, create a new window using the **New...** command of the **Windows** menu (we shall call this window **Mymenus**). When the window initially appears it will be on top of your first window, but you can resize it and / or move it so you can see both windows at once. However, only one window is the 'front' window at any one time - the one with the horizontal stripes across the title bar, and the one you are typing into.

Type into the **Mymenus** window the program shown below, and add some more facts to the **Test Window** window.



`setof`, `install_menu` and `message` are all MacPROLOG primitives - see the Reference Manual for further details.

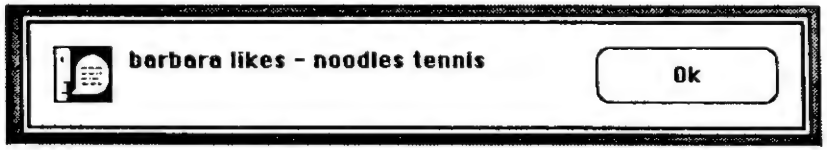
Now enter the query

```
whomenu
```

(Don't forget to either select the **Check program** box in the **Query...** dialogue, or do a **Check program** from the **Eval** menu before running the query.)

You will see a new menu **Who** appear in the menu bar. The `'Who'` program we have written will be called whenever you select an item from that menu. (See the chapter on Menu Handling in the Reference Manual). Try it.

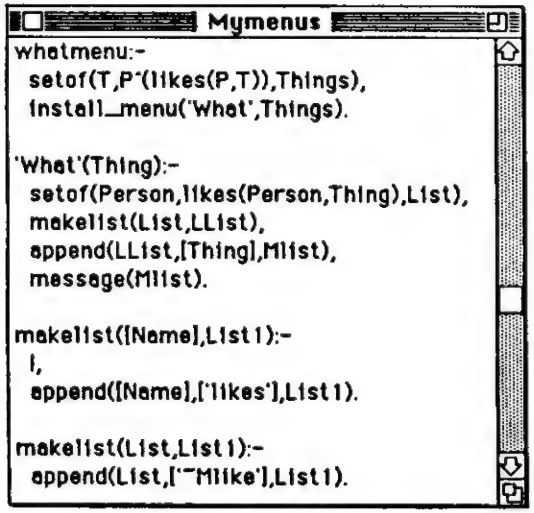
For example, if you select **barbara** from the menu, a dialogue will appear at the bottom of the screen:



This dialogue is generated by the message primitive.

Notice that the call to `install_menu` automatically picks up all the 'people' from our data base. If we change the data base, say by adding another `likes` clause, and then run whomenu again, the new **Who** menu will reflect this new state of the data base. (It doesn't matter that a **Who** menu already exists - the `install_menu` primitive automatically deletes the old one before installing the new one.)

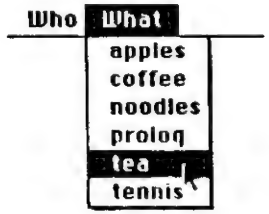
In a similar way, we can create a **What** menu whose items are all the 'liked' things of our data base. Type into your **Mymenus** window the whatmenu program shown below.



Enter the query whatmenu. A new menu **What** will appear on the menu bar.

The program for the menu items is similar to the one for the **Who** menu, except that we do a bit of list manipulation. In the display dialogue, if only one person likes something, we use the word `likes`, and otherwise we use `like`. Note also that we have used '~M' to denote a carriage return, so that `like` will appear on a new line.

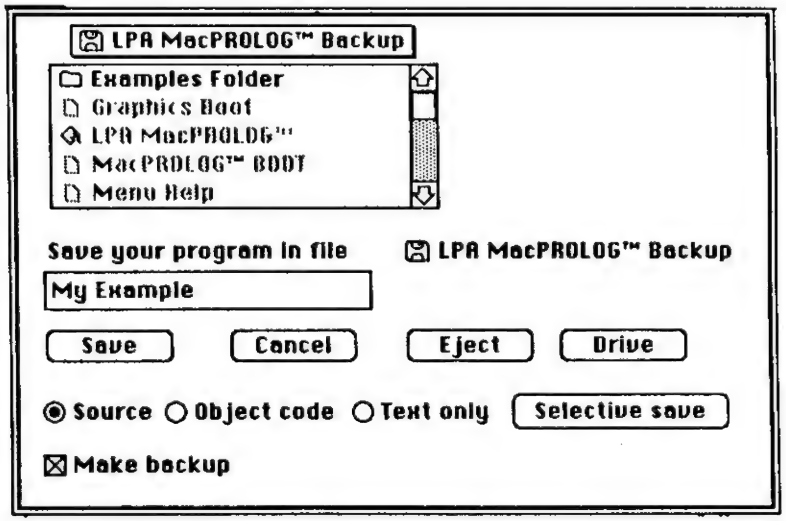
Again, experiment with this new menu. For example, if you select **tea** from the menu:



you will be told that

5.6 Saving a program

At this stage you may like to save the program you have just written. Select the **Save...** option from the **File** menu. You will see a **Save...** dialogue something like this.



The file name field will always show the name of the file last mentioned in a **Load...** or **Save...** dialogue. File names may contain spaces to make them more readable.

Select the disk and/or folder where you want to save this file, and type in a suitable file name - we shall use the name **My Example**.

The three radio buttons labelled **Source**, **Object code** and **Text only** correspond to three different forms of saved program.

Select the **Source** option (it will probably already be the one selected) and click on the **Save** button. This saves your program as a collection of windows (two in this case), so that when you load it again from disk the program windows will reappear on the screen exactly as you left them. The windows will also all be compiled as they are loaded.

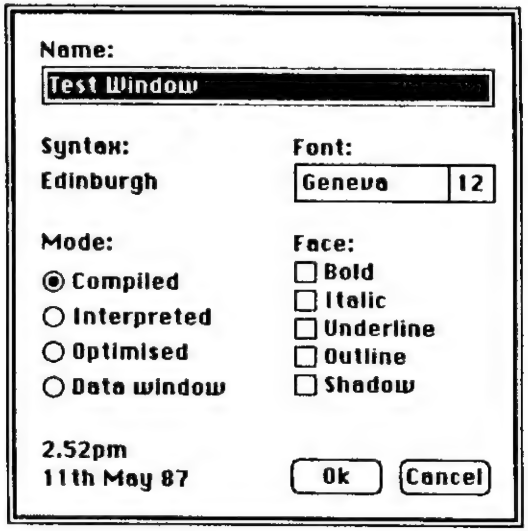
5.7 Interpreted Windows

The two edit windows we have created so far have been *compiled* windows. Programs in compiled windows are compiled a relation at a time, producing a contiguous block of code for the entire relation. Consequently a running program can't change the definition of such a relation using `assert` and `retract`. We sometimes refer to such programs as static.

To be able to side-effect a relation during an evaluation requires that each clause be separately compiled and then loosely linked to the other clauses for the relation. In MacPROLOG programs compiled a clause at a time are called *interpreted* or *dynamic* programs. The PROLOG data base primitives such as `assert`, `retract` and `clause` apply only to interpreted programs.

We can specify that the relations defined in a given edit window should be treated as dynamic, interpreted programs by changing the compilation mode of the window.

Make **Test Window** the front window. Select the **Window details...** option from the **Windows** menu. You will see a dialogue like this.



The **Mode** of the window is **Compiled**. This is the default mode when a new window is created (although you can change this default, as explained later). Select the **Data window** radio button and click **Ok**. (If you wish, you can also rename the window here, or change its text font, style or size.)

A *Data* window is a special form of interpreted window. When we add to or delete clauses from a relation in a Data window, the text of a Data window is automatically updated to reflect these changes. The programs in an **Interpreted** window are also dynamic but the text of the window is not updated after an `assert` or `retract`.

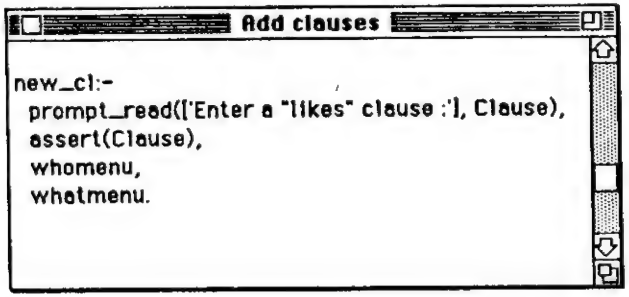
You can see this effect immediately if you enter the query

```
assert(likes(frank,tennis))
```

and look at the text of **Test Window**. To delete this clause again enter the query

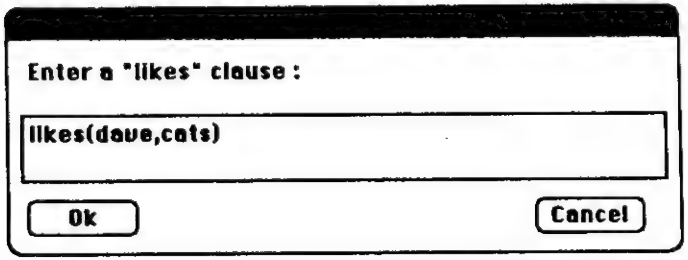
```
retract(likes(frank,tennis))
```

To get a new clause from a user, we will use the primitive `prompt_read`. Create a new window (we'll use the name **Add clauses**) and type in the program `new_cl`.



The MacPROLOG primitives `prompt_read` and `assert` are described in the Reference Manual.

Enter the query `new_cl`. You will get a dialogue (generated by the `prompt_read` primitive) into which you may type a new clause. Suppose you type



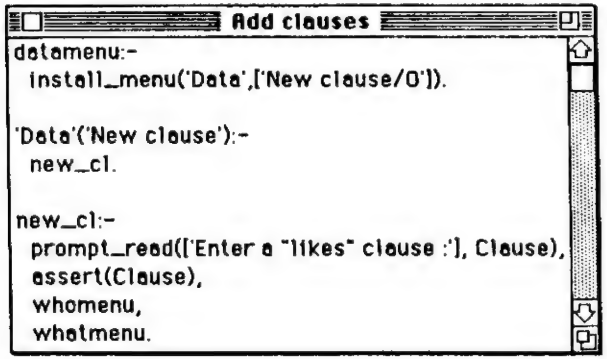
and click on the **Ok** button. You will find that the following things have happened:

a) the **Who** menu now contains **dave**
b) the **What** menu now contains **cats**
c) the **Test Window** window now contains the clause `likes(dave,cats).`

The menus are updated by the calls to our programs `whomenu` and `whatmenu`; the **Test Window** window (and the data base) is updated by the call to `assert`.

(Note that if we had made **Test Window** an interpreted window, we would see no textual change to the window even though the clause would still have been added to the data base.)

A good way of offering the above dialogue to a user would be to use a menu again. We'll add another one-item menu by calling `install_menu`.



Calling `datamenu` will add this new menu. Note the `/O` following the menu item - this means that the menu item can be selected from the keyboard by pressing the 'Command' key (⌘) and O together. You will see a **⌘O** next to the menu item to indicate this.

You could if you prefer simply rename the `new_cl` program to `'Data'('New clause').`

5.8 Hiding, Showing and Killing Windows

We will now take a short break from programming to explore some of the window handling facilities of the MacPROLOG programming environment.

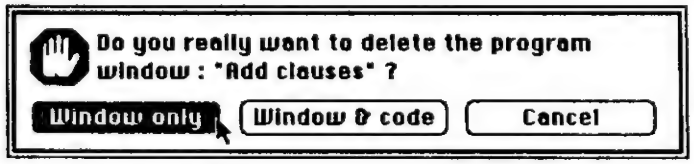
Before we go any further, save your program again. As before, select the **Source** option and click on the **Save** button. If you are saving under the same file name you will be asked if you want to replace the previous version. If the **Make backup** box is selected the previous version will be copied before saving this new copy anyway.

Select **Hide** from the **Windows** menu. The front window becomes invisible. (This has the same effect as clicking in the window's *goaway* box in the top left hand corner of the window.) Unlike many other Macintosh applications, just because the window is invisible does not mean its contents have been deleted. Select **Select window...** and you will be shown a scrolling menu of the available windows, and the window you have just hidden will be among them. You can select it from here and it will become visible again.

If you select **Hide all** from the **Windows** menu all your windows will disappear. You can bring them all back again by selecting **Show all**. Alternatively you can choose one or more windows from the scrolling menu in the **Select window...** option.

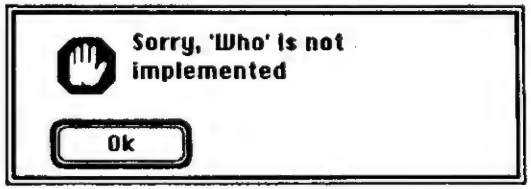
Select the **Clean up** option from the **Windows** menu. You will find that this has the effect of neatly stacking all your windows on the screen. The order in which they are stacked is alphabetical (by window name) if the **Alphabetical** menu item is selected (you will see a tick beside the item); otherwise they are stacked in date order. Windows of any given type are grouped together. The **Default Output Window** is a display window and is therefore stacked separately from your program windows. If you later create any graphics windows they will be grouped together.

To permanently remove a window you must kill it. Make the **Add clauses** window the front window, and select **Kill** from the **Windows** menu. You will be offered the options of deleting either the window only, or the window and the code.



Click the **Window only** button. The window (and its source text) is now lost, but the code produced from compiling the window is still there. You can check this by selecting **New clause** from your **Data** menu - the program defining this menu item (which was in the window we've just killed) still works!

Now make **Mymenus** the front window, select the **Kill** option and this time delete the window *and* the code. Try selecting an item from the **Who** or **What** menu. Sadly, you will be told

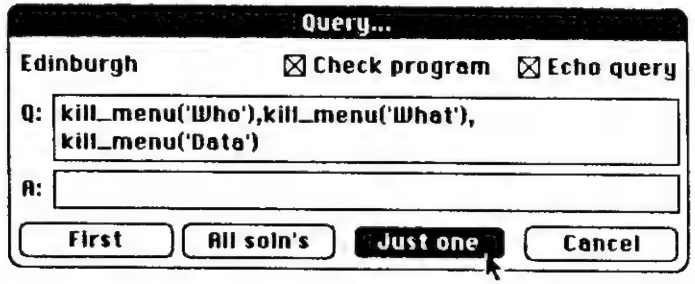


This time the window's code has certainly gone too . However, all is not lost because we saved all our work before doing any of this.

5.9 Reinitialising

We will reload the program, but first we will completely clear the Mac's memory. Select the **Reinitialise...** command from the **File** menu. This is like a "Kill all". If you choose to delete **Windows & code** then this effectively puts MacPROLOG back into its initial state, as it is immediately after booting up. Do this.

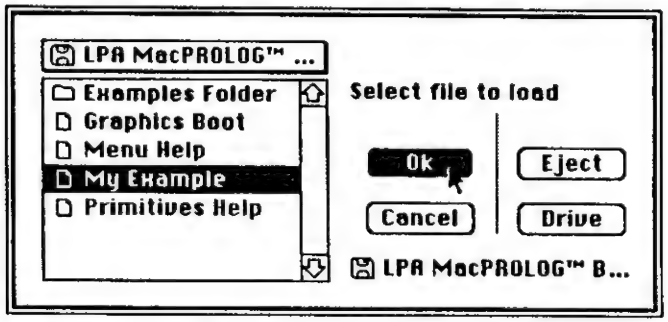
There is one last thing to deal with, and that's the (now non-functioning) menus **Who**, **What** and **Data**. We can dispose of them by entering the following query



where `kill_menu` is a MacPROLOG primitive which deletes a menu.

5.10 Loading a PROLOG program

Having cleared our program completely we'll now load it in again. Select the **Load...** option from the **File** menu to get a standard Macintosh **Load** dialogue, something like this.



Select the **My Example** program (or whatever you called the file) and load it. MacPROLOG automatically detects that it is a source program (saved with the **Source** option) and loads it accordingly.

Because the file was saved as source, the windows come back as they were when they were saved, i.e. with the same names, contents, positions, compilation modes, fonts etc. They are also compiled on loading so you can straight away run the query

```
whomenu, whatmenu, datamenu
```

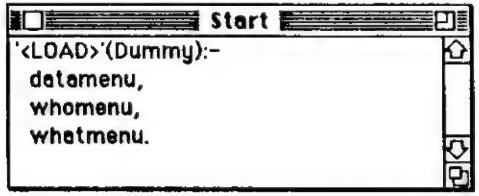
to install the **Who**, **What** and **Data** menus again.

5.11 The "Execute on Load" facility

Instead of having to run the queries `whomenu` and `whatmenu` after loading, the menu installation can be made automatic on loading. Obviously this is preferable if this is to be an application directed at naive users.

All that is needed is a definition for the relation `'<LOAD>'`. This has to be a unary relation, but the single argument is ignored.

Create a new window (here it's called **Start**) and type in the following definition for `'<LOAD>'`.



Compile the window with the **Check program** command. Now do a **Save...** again (and use a different file name this time).

To see this working, **Reinitialise...** to remove all the existing program code, and kill the **Who**, **What** and **Data** menus if they are there.

Now **Load...** the program you have just saved. You will see that as soon as the load is complete, the three new menus appear in the menu bar. And of course they contain whatever items are in the data base (as displayed in the **Test Window** data window) at the time of loading.

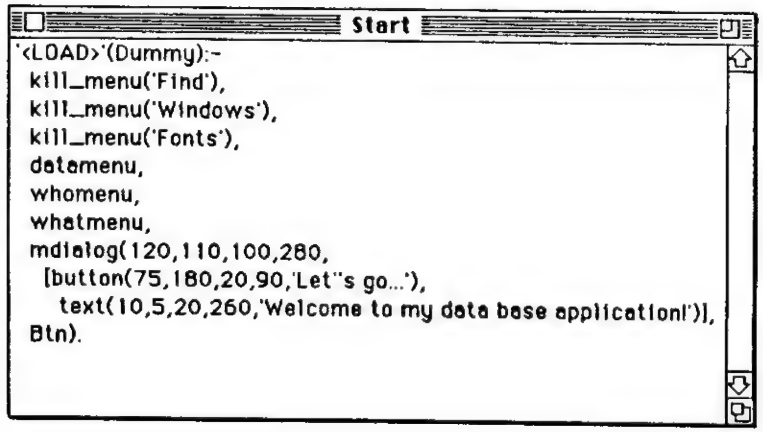
(See the chapter on Implementing an Application in the Reference Manual for further discussion of the `'<LOAD>'` program.)

5.12 Adding some Frills

You may well want to remove the menus of the programming environment before installing your own menus. This can be done very simply by making a few calls to `kill_menu` in the `'<LOAD>'` program.

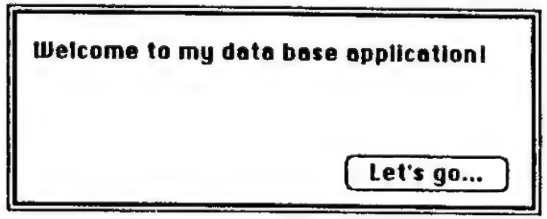
Also you might like to have a 'Welcome' message appear on the screen when your application is started. This could be done with a call to the MacPROLOG primitive `mdialog` (described in the chapter on Dialogue Building in the Reference Manual).

The following amended `'<LOAD>'` program will do these things.



(You could also kill the **Eval** menu too, but you should not kill the **File** menu.)

The call to `mdialog` will present the dialogue



to which the user must respond by clicking on the **Let's go...** button.

WARNING If you type in this program, save it, and load it again, you will lose the **Find**, **Windows** and **Fonts** menus! The simplest way to get back to the full MacPROLOG programming environment is to **Quit** and re-load MacPROLOG.

5.13 Some Comments on the Example Program

In our example data base program we used the menus of the menu bar in a slightly unconventional way, but it served to demonstrate the ease with which menus can be created, implemented and altered. In general you will use the menu bar to offer commands to the user rather than using them for data display purposes. There is only room for a limited number of menus on the menu bar, so to take up two of them for the **Who** and **What** options is rather inappropriate.

You can offer a menu to a user in a different way, using the `scroll_menu` primitive of MacPROLOG. This presents a scrolling menu inside a dialogue. So for our data base program we could instead install just one new menu in the menu bar which contains the three commmands **Who**, **What** and **New clause**, and generate a scrolling menu for the **Who** and **What** options.

Some of the amended program might look like this. (See the Reference Manual for descriptions of the MacPROLOG primitives `scroll_menu` and `map`.)

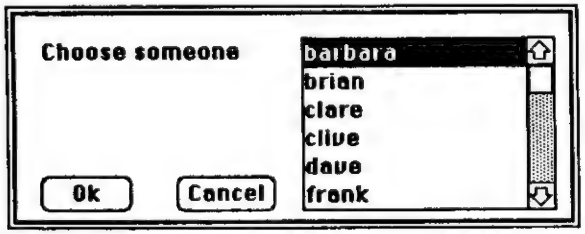
```
newmenu:-
        install_menu('Likes',['Who','What','(-','New clause...']).

'Likes'('Who'):-
        setof(P,T^(likes(P,T)),Them),
        Them=[First|Rest],
        scroll_menu(['Choose someone'],Them,[First],Selection),
        map(who,Selection).

who(Person):-
        setof(Thing,likes(Person,Thing),List),
        message([Person,'likes -' | List]).
```

The effect of the `'(-'` in the item list for `install_menu` above is to insert a separating bar between the items.

The `scroll_menu` will display a dialogue as follows



By holding the shift key down whilst clicking, you can select more than one item from this menu. The list of items selected is returned in the `Selection` variable above, and we generate a `message` dialogue for each 'person' on this list, using `map`.

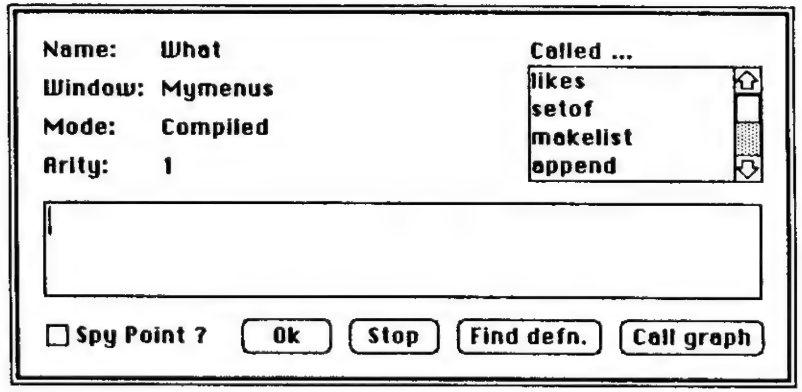
The **What** menu item could be dealt with similarly.

6. Further features of the programming environment

6.1 Information about relations

The environment offers some very powerful text searching facilities. In the **Find** menu are the usual "Find and replace" text options common to most Macintosh applications. However, there are further options which are special to MacPROLOG.

First we will look at the **Get information...** option in the **Find** menu. Hide all your windows and select **Get information...**. You will be presented with a scrolling menu of all the currently defined relations. Choose one of these, say `'What'`. You will then see a dialogue giving information about the relation `'What'`.



The information that is displayed is the name of the window in which the relation is defined, its compilation mode, its arity or arities (if the relation is defined by clauses with different numbers of arguments), and also a list of all the relations that are called by `'What'`.

The large empty box above the row of buttons is an edit field into which you may type any comments about the relation. These comments will be remembered as assertions for the environment `'COMMENT'` relation and will be displayed next time you do **Get information...** on `'What'`. All comments are also stored in a **<Comments>** window. You can make the **<Comments>** window visible using the **Select window...** command, and you can change the comments by editing the text of this window.

If you now click on the **Find defn.** button then the program window in which the relation is defined is brought to the front (and made visible if necessary), and the first occurrence of the relation name in the window, which is usually the head of the first clause for that relation, is highlighted.

You can also find the definition of a relation by selecting **Find definition...** from the **Find** menu.

There is another way to generate this dialogue. When you select **Get information...** , if there is a relation name currently highlighted in the front window, the information will automatically be generated for that relation. Alternatively, the cursor may be either in or next to the relation name (the whole name does not actually have to be selected). For example, if you now select **Get information...** (still with `'What'` highlighted in the front window) then you will not get a scrolling menu : instead the above dialogue will be displayed immediately.

A similar procedure operates for the **Find definition...** option.

Both of these menu items determine the relation to operate on by first seeing if a relation name is selected in the text of the front window, then searching for a relation name next to the cursor in the front window. Finally, if neither of these procedures yield a relation name, the scrolling menu of names will be presented (as in our first use of **Get information...** described above).

Experiment with these two facilities - they become extremely useful when your programs start to get large. They enable you to go straight to the definition of a relation even if it is defined in a window that is currently invisible. Note, too, that you can use the key combinations **⌘D** or **⌘I** as a shortcut to these menu items.

6.2 Checking brackets

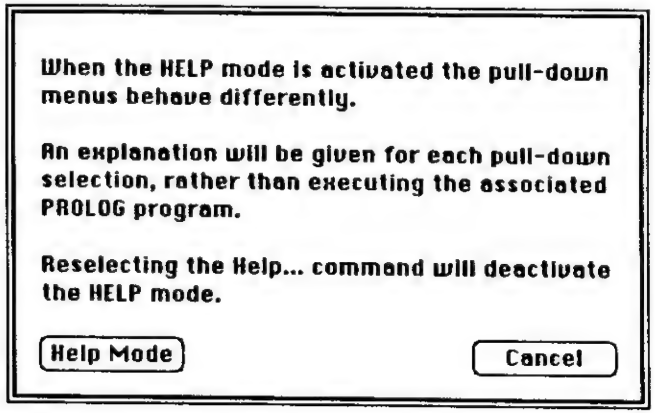
There are often times when programming in MacPROLOG that you lose track of the brackets in an expression. The **Balance** command of the **Edit** menu is designed to help you pair up matching brackets. Place the cursor inside some pair of brackets and select **Balance**. You will see that a section of text around the cursor becomes highlighted - this represents the contents of the nearest matching pair of brackets. If you select **Balance** again then the highlighting will be extended to include the next pair of matching brackets, and so on. If you have inadvertently 'lost' a bracket somewhere then this should soon become clear!

(Note : There is one circumstance where **Balance** will not work properly. That is when you have a bracket character inside some quoted atom, which will be treated as a normal bracket and will probably cause incorrect matching.)

6.3 Getting Help

There is a help facility available which enables you to get a brief description of many of the menu commands from within the programming environment. To do this you must select 'Help mode'.

Select the **Help...** option from the **File** menu. A dialogue appears telling you about the Help mode.



Click on the **Help Mode** button. The **Help...** option now has a tick beside it in the **File** menu. Now, whenever you select a command from the **Edit**, **Find**, **Windows** or **Eval** menus, instead of that command being executed, you will get a dialogue giving you information on what that command does.

Notice that the first time you ask for help on any menu item there will be short pause while the help file is loaded from disk. You will see a dialogue box telling you that the help file is being loaded.

To turn the help mode off just select **Help...** again.

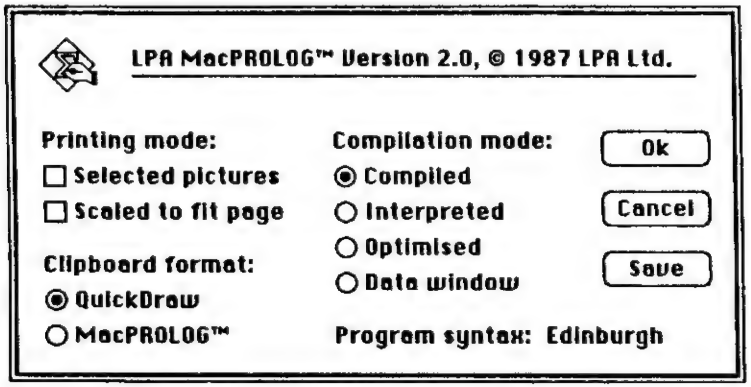
Note: The help file is simply a text file containing information on the menu items. When it is loaded its text is stored in an invisible display window. If you wish, you can look at this window after it has been loaded by choosing the **Menu Help** window from the **Select window...** option of the **Windows** menu (with Help mode off!). However, because it is a *display* window (similar to the **Default Output Window**) you cannot type into it. Select **Hide** to make the window invisible again.

You can of course edit the help file using MacWrite or some other editor. You can also create similar help files of your own - see the description of the `help` primitive in the Reference Manual for further details.

6.4 Setting Defaults

In the data base program example, whenever we created a new program window it was automatically created as a *compiled* mode window. You can set the default compilation mode of new windows to another option using the **Defaults...** option in the **File** menu. However, as shown earlier, whatever the default mode is, the compilation mode of a window can always be changed after it has been created, using the **Window details...** option.

Select **Defaults...** You will see a dialogue like this.



If you change the **Compilation mode** here then whenever you create a new window using **New...** it will be in the mode you have specified.

In the **Defaults...** dialogue, if you click on the **Ok** button then the defaults you have set will apply only for the duration of this session (i.e. until you exit from MacPROLOG). If, however, you click the **Save** button then the defaults you have set will be saved to the LPA MacPROLOG™ application file and will therefore be remembered for the next time you use MacPROLOG.

The **Printing mode** and **Clipboard format** are only applicable if you have the MacPROLOG™ Graphics package (described later).

7. Hints on window management and program structure.

The architecture of the MacPROLOG™ system is radically different from most other implementations of PROLOG, and indeed from other Macintosh applications. The multi-window environment may at first be confusing, and so we give a few helpful suggestions to get you started.

7.1 Relation definitions

There are only two restrictions on the way that you can enter relation definitions in windows within MacPROLOG. These are:

(i) A relation cannot be defined in more than one window.
(ii) The clauses for a relation must be contiguous within a window.

This means, for example, that if you enter

```
likes(nicky,grapes).

is_liked(Item):-likes(Someone,Item).

likes(anna,chess).
```

in a window then the compiler will generate an error because the two `likes` clauses are not contiguous. The `is_liked` clause must be placed either before or after all the `likes` clauses, or placed in a different window.

The restriction applies even if the clauses have different arities. So a clause with two arguments and a clause with three arguments which share the same relation name must be defined contiguously. However, for a given relation name, the clauses may be defined in any order within that relation's definition, regardless of their arity. MacPROLOG does not distinguish between clauses with the same name but differing arities.

7.2 How many windows should I use ?

You have seen in the example we have just done that we used several different windows to write our program. We didn't have to do it that way . We could have used just two - one for the compiled programs and one for the data base facts. However, it is more effective in terms of program management to split up a program into several different windows. It is sensible, for example, to have one window where all the menu implementation programs are defined, one window for the start-up procedures and so on.

The most important reason for using multiple edit windows is that they are incrementally compiled. That is, whenever you invoke the compiler only those windows which have been changed are recompiled. Windows which have not changed will not be recompiled. If you are working on one particular part of your program you will not have to keep recompiling *all* your source each time you make a change. If you have typed all your source into a single edit window then recompilation may be a time-consuming process.

However, at the other extreme, we do not recommend that you have a separate window for every relation. There is a memory overhead for each window, and the screen can become irritatingly cluttered if you have too many windows! We recommend an approach somewhere in the middle whereby the source text is partitioned amongst several, relatively small windows. Wherever possible, relations which are logically connected to one another should reside in the same window.

The amount of text that can be stored in an edit window in MacPROLOG is 32K. If you exceed this limit, for example by typing or pasting into a full window, you will be told that the window's text buffer is full. However, if you have followed these guidelines on window management you should never even approach this limit!

7.3 Hide your windows

Unlike many other applications, MacPROLOG™ source windows can be either hidden or visible. Even when they are hidden they will still be recompiled, searched through, and saved just as if they were visible. Whether hidden from view or on the screen, all windows are active until they are explicitly killed.

This gives you the ability to hide windows which are currently not being changed or accessed. Hiding windows whenever possible is a recommended technique for keeping the screen tidy.

7.4 The Default Output Window

You can always scroll through the text of the **Default Output Window** to refer back to the results of some previous query or to examine the results of a trace, for example. However, all the text is occupying memory in the computer and it is a good idea to clear this memory from time to time, particularly if you are getting 'Out of Memory' errors. You can either **Clear** any parts of the text you no longer need, or you can clear the entire window by bringing the window to the front and doing a **Select all** followed by a **Clear** from the **Edit** menu.

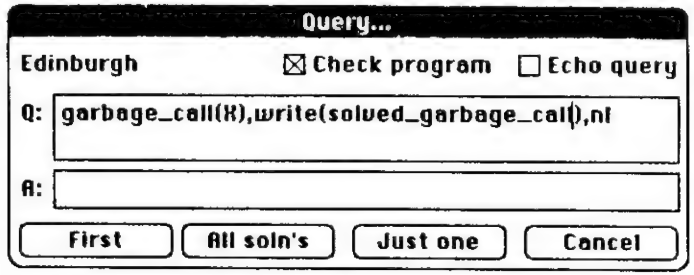
7.5 Other types of window

The **New...** command of the **Windows** menu always creates a program edit window. Its compilation mode will be whatever has been set as the default using the **Defaults...** option of the **File** menu. If you want to create windows of other types (for example the **Default Output Window** is a *display* window) then you must call one of the MacPROLOG window creation primitives `wcreate`, `wpcreate` or `wgcreate` - see the Reference Manual for a description of these.

8. Errors

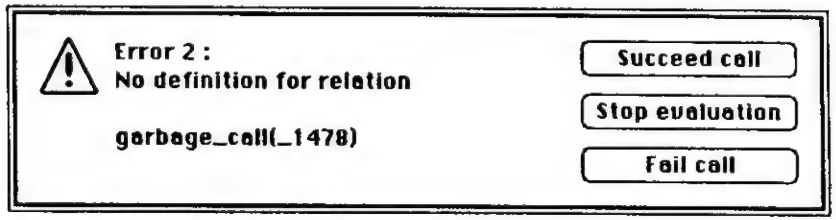
8.1 Default Error Handler

Sometimes you will make mistakes and mistype something, or forget to define a relation before trying to use it. Errors such as these are picked up by the default MacPROLOG error handler and reported to you using an error dialogue.



Try running the above query.

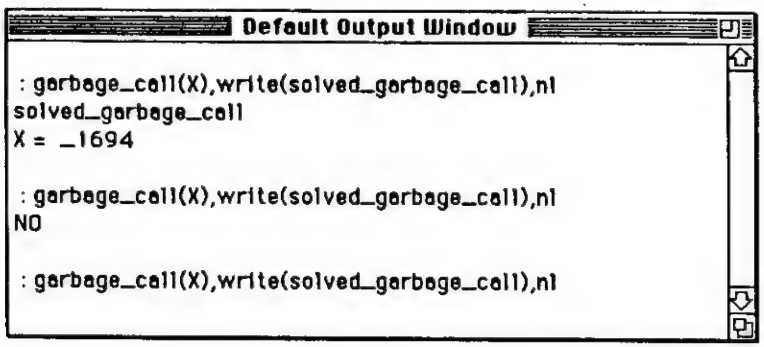
MacPROLOG will display an error dialogue warning you that it does not know about the relation `garbage_call`. The dialogue has several buttons which allow the error to be handled in different ways. Displayed in the dialogue box is the call that caused the error.



Selecting **Succeed call** will cause the execution to proceed as if the call had succeeded (without binding any variables in the call). If you select this option then the other call in the query, the call to `write`, will be executed. Click the **Succeed call** button to see this.

The **Fail call** button causes the execution to backtrack as if the call itself had failed. This will prevent the `write` from being executed and the query will terminate. Run the query again but this time click the **Fail call** button.

The **Stop evaluation** button is used to stop the current query altogether. The `write` will never be executed. Run the query again but this time click the **Stop evaluation** button .



You should see the above output in the **Default Output Window** produced by the three different responses: **Succeed call, Fail call** and **Stop evaluation**.

You can define your own error handler. For details, see the chapter on Implementing an Application in the MacPROLOG Reference Manual.

You can also use `dynamic` to prevent the error handler being called when there are no clauses for a particular relation. See the description of `dynamic` in the Reference Manual.

8.2 Errors When Using Primitives

You will sometimes get the error "Invalid form of use" when calling a MacPROLOG primitive. It either means that you have the wrong number of arguments or that one or more arguments are of the wrong type.

When you get an error related to the use of a system primitive you may find that the name of the primitive given in the error dialogue is different from the name of the primitive appearing in your program. Often the name displayed will be an upper case variant of the name that you have used. This is because MacPROLOG version 2.0 allows upper case synonyms for many of its primitive relations, for compatibility with MacPROLOG version 1.0.

You can see all these upper case synonyms, and all the other reserved names of MacPROLOG, by running the query

```
sdict(X).
```

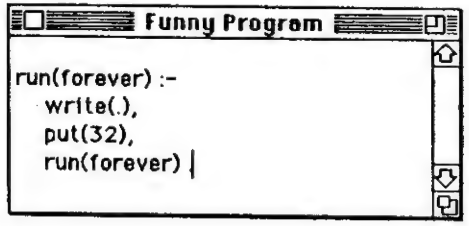
This will bind `X` to a list of all the names of the MacPROLOG system relations. If you try to redefine any of these reserved relations you will get the error message

```
Cannot redefine sdict relation :  ...
```

and the compilation of the window containing the definition will be aborted.

8.3 Halting a program

Sometimes you will find that a program you have defined will go into a never ending loop and that you will need to stop it.



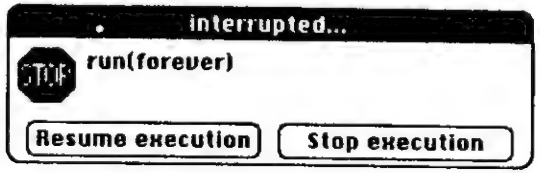
Create a new window, (we shall call it **Funny Program**), containing the program as shown above. This just repeatedly writes dots to the **Default Output Window**!

A MacPROLOG program can always be stopped by pressing the full stop key with the ⌘ key held down. This will produce a special dialogue asking if you want to halt the current execution or continue as if nothing had happened.

Run the query

```
run(forever).
```

Press ⌘. to stop the execution of this non-terminating program. You will see the following dialogue on the screen.

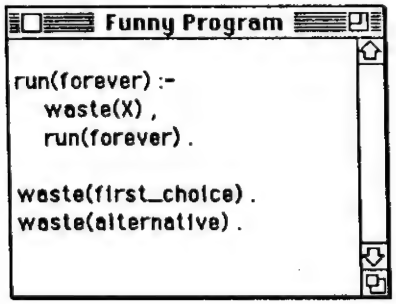


Click the **Stop execution** button to stop the program running.

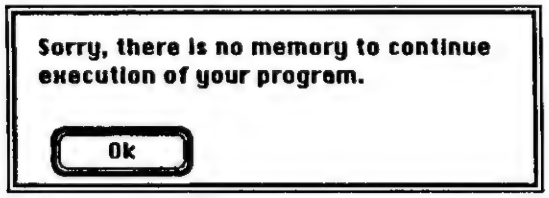
NOTE: it is not possible to interrupt a **LOAD...** sequence.

8.4 Running out of memory

It is possible to define programs that consume memory as they run. Eventually MacPROLOG will run out of memory and produce an error message. The following modification of the `run` definition will use up stack space as it executes because it is no longer tail recursive.



Alter the previous program to the above definition and run the query `run(forever)` again. Soon you will get the message



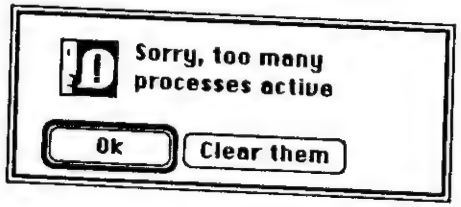
Kill the **Funny Program** window and the code for its programs.

Another, quite different way in which you can run out of memory is by having too many active processes. The MacPROLOG architecture includes two separate evaluation spaces, referred to as the primary and secondary stacks. When both these stacks are in use no new evaluations may be started.

A MacPROLOG process suspends when it displays a modeless dialogue. At that point you can start another MacPROLOG evaluation instead of responding to the dialogue. However, because there can only be two current MacPROLOG processes, you can only do this once.

Try selecting the **What to find...** option from the **Find** menu. Don't respond to it, but select **New...** from the **Windows** menu. Again, leave this dialogue where it is without responding. Now try to do something else - for example, try to enter a query.

You will be beeped at and told



First, click on the **Ok** button. The effect of this is just to return you to where you were. You must respond to at least one of the current dialogues before you can continue. If you don't, you will just keep getting the above error dialogue back again!

Now create the same situation again. This time click the **Clear them** button. This will clear the currently active processes. In this case the **Find/replace** and **New window...** dialogues will disappear as if you had clicked the **Cancel** button on each. Finally the **Query** dialogue (or whatever you had tried to do) will appear and you may now continue.

9. Parsing Example

We will now look at a slightly more complex example of a MacPROLOG program, and use it to investigate the debugging and tracing facilities available in MacPROLOG. In the **Examples Folder** on the LPA MacPROLOG disk you will find a file called **Guide Example**. Load this file.

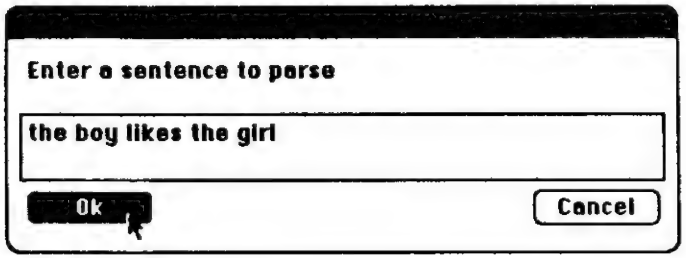
It has been saved as source text so you will see the program windows appearing as it is loading. They are also being compiled as they are being loaded.

This program is an elementary parser for English sentences. It asks the user to type in a simple sentence such as

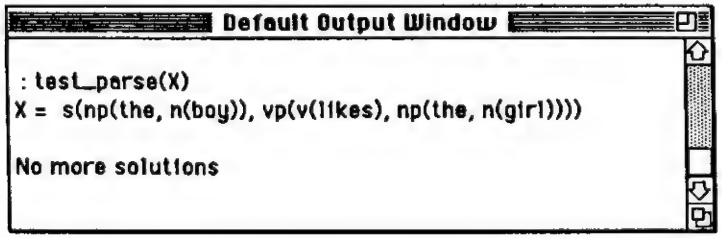
```
the large dog bit the boy
```

and the program then parses it into its grammatical structure.

The **Parser Test** window explains how to invoke the parser. If you parse the sentence



then you will get the answer given below.



The term structure is explained in the **Parsing Example** window.

If you enter a sentence containing a word that the parser does not 'recognise', the query `test_parse(X)` will simply fail. However, you can add to the program's vocabulary by entering extra clauses into the **Dictionary** and **Small Words...** windows.

10. Debugging with Spy Points

It is important, when developing a program, to be able to examine how it executes. Tracing in MacPROLOG occurs at source level making it much easier to debug programs.

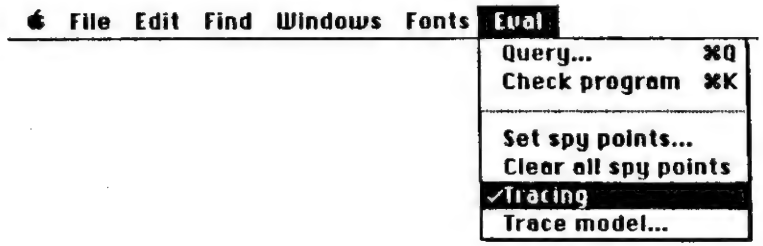
Spy point tracing is the simplest form of debugging, and consists of monitoring the entry/exit behaviour of programs for specified relations.

We assume here that you still have the Parsing Example program loaded in MacPROLOG.

10.1 Enabling tracing

The first thing to do when debugging a program is to turn the tracing facility on. Tracing is enabled when the **Tracing** item in the **Eval** menu is ticked, and is disabled otherwise. Selecting this item will toggle between the two modes.

Make sure there is a tick next to the **Tracing** command in the **Eval** menu.



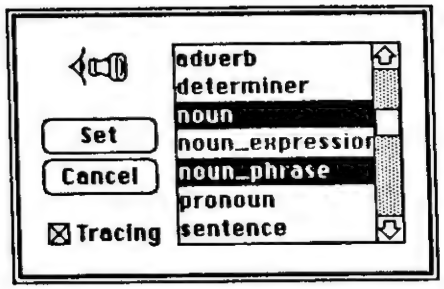
NOTE: You will only get debugging information when tracing is enabled.

10.2 Setting spy points

Using the **Set spy points...** command of the **Eval** menu spy points can be set for any compiled or interpreted relation. You will be presented with a scrolling menu of all known relations, from which you can then select (or deselect) the spy points.

Select the **Set spy points ...** command from the **Eval** menu. Click the menu entry for `noun` and then shift-click on the entry for `noun_phrase`. Click the **Set** button to set the spy points on the indicated relations.

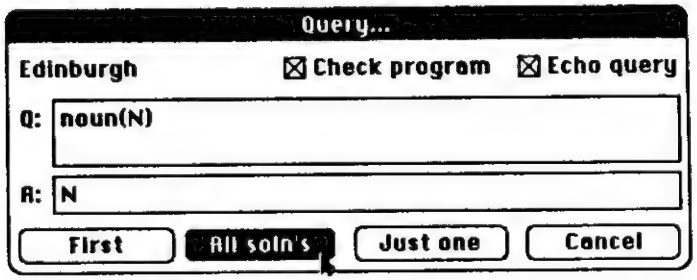
Note that the **Tracing** check box is on. Setting the check box here is an alternative to selecting the **Tracing** item of the **Eval** menu.



(An alternative, and sometimes quicker, way to set a spy point on a relation is to use the **Get information...** command on the relation, and then click on the **Spy point ?** check box in the information dialogue.)

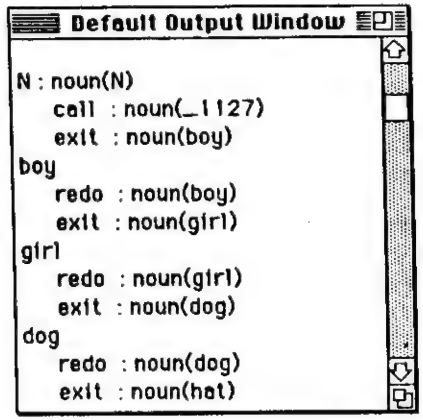
10.3 Spy point output

We will enter a query to find all the nouns currently defined. Type it in as follows.



Click on the **All soln's** button. The results will be displayed in the **Default Output Window**.

Remember that after the query has finished you can use the scroll controls to move up and down the output shown in the window.



As you will see there are four types of spy point messages : **call, exit, fail** and **redo.**

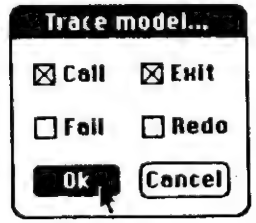
The **call** message is output whenever a call is made to a spied relation. If the call subsequently succeeds an **exit** message is displayed. Notice that at an **exit** port some arguments may have a value where previously (i.e. at the **call** port) there was a variable. If the call does not succeed, a **fail** message is displayed. If the call succeeds, but at a later stage an alternative solution is required, then MacPROLOG will backtrack to try and find a different way to solve the call. This is indicated by a **redo** message.

10.4 Selective output

The above query resulted in all spy point information (**call, exit, redo** and **fail**) being displayed for the noun program. This can sometimes result in a profusion of messages which is then hard to interpret. Very often, in the debugging process, we are looking for a particular behaviour.

The **Trace model...** command from the **Eval** menu allows you to specify which of the spy ports (**call, exit, redo** and **fail**) will actually generate messages. For instance, if some program is unexpectedly failing then there may be no need to see the **call, exit** or **redo** ports, only the **fail** port.

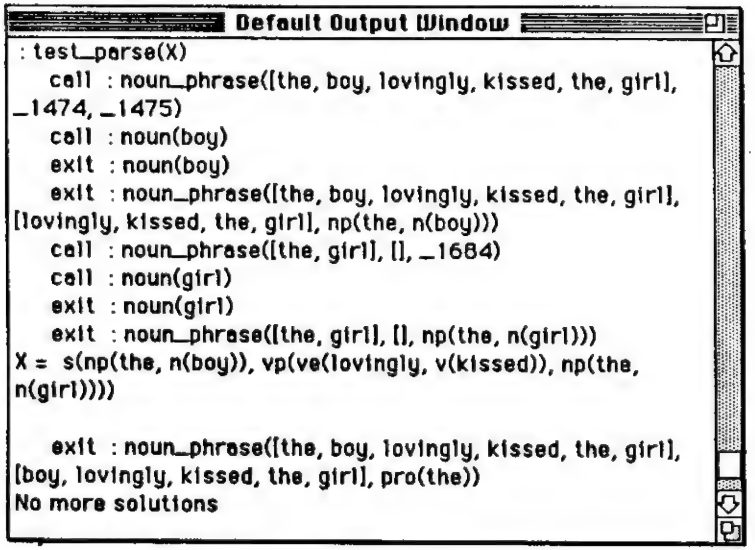
Select the **Trace model...** command from the **Eval** menu and select only the **Call** and **Exit** options from the dialogue.



Run the query `test_parse(X)` again, choosing the **All soln's** button, and type

```
the boy lovingly kissed the girl
```

as the sentence to parse.



This time you will only see the **call** and **exit** messages.

10.5 Removing spy points

The **Clear all spy points** command of the **Eval** menu will remove all spy points which have previously been set. Programs will now execute with no debugging information.

To remove spy points from individual relations, select the **Set spy points...** option again and deselect the relations from the scrolling menu for which you want to remove spy points.

(Note that you can temporarily prevent tracing information being displayed by deselecting **Tracing** in the **Eval** menu. The spy points still remain set but will only be 'active' when **Tracing** is enabled.)

11. Full tracing

Spy point tracing only provides a certain level of trace information. More information can be gained by doing a full trace of a program execution. For full tracing the relation must be an *interpreted* relation.

To get full trace information about the `noun_phrase` program we must change the mode of the window in which its source resides from Compiled to Interpreted (or Data).

Bring the **Parsing Example** window to the front and then select the **Window details...** command from the **Windows** menu. Change the compilation mode to **Interpreted** and then select the **Check program** command from the **Eval** menu to force a recompilation of the **Parsing Example** window.

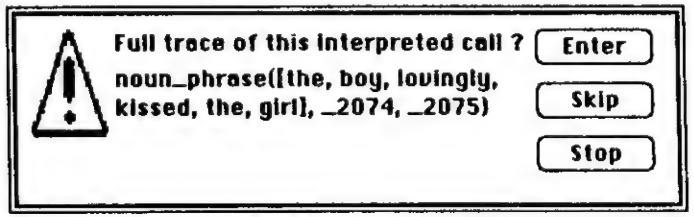
Finally, if you previously cleared all the spy points, reset the spy points for `noun` and `noun_phrase`.

11.1 Controlling the trace of a program

Re-run the last query using the same sentence `the boy lovingly kissed the girl`.

Each time an interpreted spied relation is called you will be offered the chance to follow through a full trace of its execution. If you do not want to see a full trace click the **Skip** button. This will produce no more information than an ordinary spy point trace.

The first call for which you will have the option of a full trace is the call to `noun-phrase`.



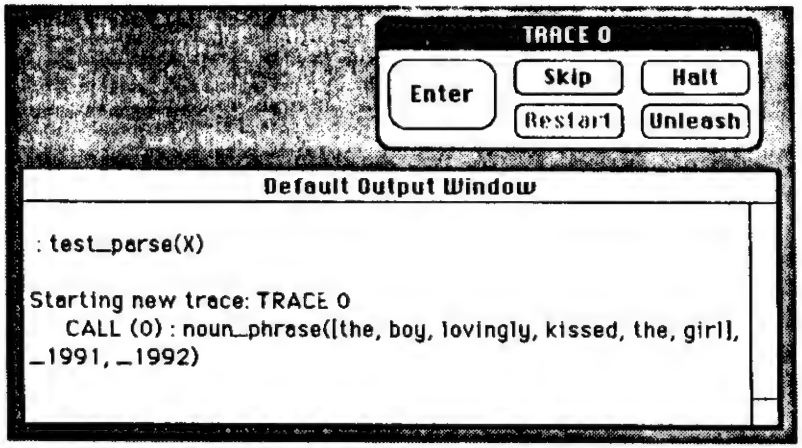
Click the **Enter** button to follow through a full trace of the call.

11.2 Single stepping a trace

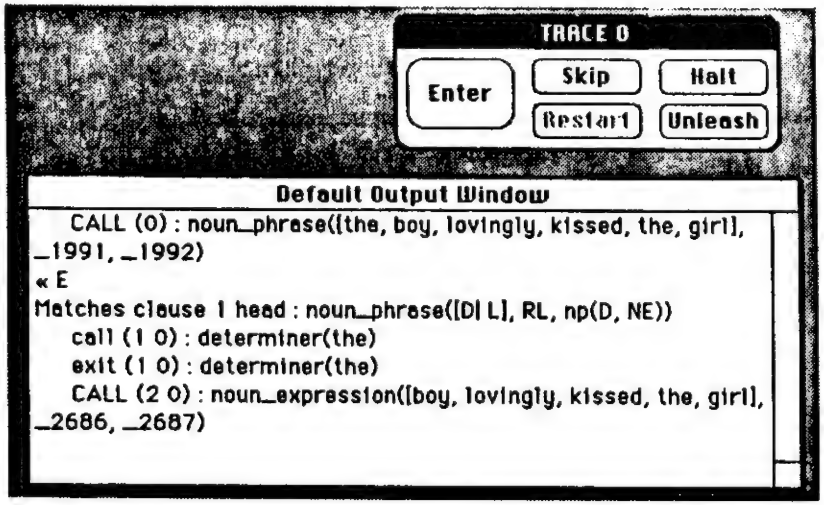
Each full trace that you do is numbered, starting from 0. The number forms part of the name of the dialogue box, named **TRACE 0**, which is used to control the various ways of stepping through the execution. To mark the start of each new full trace a message of the form

```
Starting new trace: TRACE...
```

is displayed in the **Default Output Window**. The end of each full trace is marked by displaying "`Leaving TRACE...`" in the **Default Output Window**.



Click the **Enter** button to trace through a single step of the execution.



If you single step through the execution, calls to compiled relations will be displayed without a prompt (i.e. no dialogue). For a call to an interpreted relation the trace dialogue will be redisplayed to enable you to decide upon the next action.

In our example trace, the call to `determiner` is displayed without a prompt since it is a call to a compiled relation. The call succeeds. Since `noun_expression` is an interpreted relation, you must decide whether or not to step through its evaluation.

The buttons of the trace dialogue represent various trace options. Every time you click on a button the selection is displayed in the **Default Output Window** preceded by the character «. The characters used for this display are:

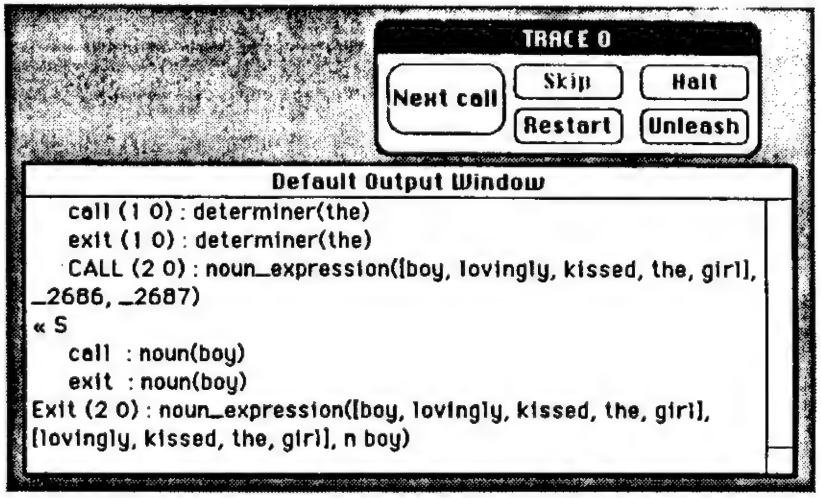
E : **Enter**
S : **Skip**
R : **Restart**
U : **Unleash**

Each call displayed during the trace is labelled with a list of numbers. The list specifies the relative position of the call within that particular trace. For example, the label (2 0) indicates the second call in the body of the clause currently being used to evaluate the initiating call for TRACE0.

11.3 Skipping a call

Now select the **Skip** button.

The **Skip** option allows you to skip over the trace details, and merely see whether the call succeeds or fails. For a skipped call only spy point information for the call will be displayed, along with spy point information for any spied relations that are called.



11.4 Re-doing a skipped call

Whether a skipped call succeeds or fails you are offered the option to restart the execution in order to see how it was solved or why it failed. The **Restart** option in the dialogue is only enabled after a **Skip**. The **Restart** unwinds the evaluation right back to the initial entry of the call. It is as though you had hit the **Enter** button instead of the **Skip** button when the call was first encountered.

Click the **Restart** button to restart the execution of the call to `noun_expression`.

This first trace will complete having shown that `"the boy"` is a noun phrase, leaving the rest of the sentence `"lovingly kissed the girl"` yet to be parsed as a verb phrase.

During the attempt to show that this expression is a verb phrase there will be a call to show that `"the girl"` is a noun phrase. Since this is a spied interpreted relation, you will again be given the option to step through the evaluation.

11.5 Unleashing a trace

Click the **Enter** button of the **TRACE** dialogue to step through the evaluation of the call to show that `"the girl"` is a noun phrase. Then click the **Unleash** button.

This time the complete trace will be output without any further interaction. The trace will show that `"the girl"` is an instance of `noun_phrase`.

Unleash causes all the trace information to be displayed without prompting, up to the next call to a spied interpreted relation. At that point you will then be asked if you want to step through the trace of that call.

12. Operators

By declaring a single or two argument functor as an operator, terms can be written without using brackets. This simplifies input and makes output more readable. For example, if `n` were declared as a prefix operator (i.e. the operator name precedes the operand) then `n (boy)` could be written as:

```
n boy
```

and if `s` were declared as an infix operator (i.e. the operator appears between its operands) then `s(NP,VP)` could be written as:

```
NP s VP
```

The mathematical operators `+ - / * ^` are examples of predeclared infix operators.

We shall declare a new operator `n` to be a prefix, non-associative operator with a precedence of 200.

Precedence indicates in which order to group the parts of an unbracketed term. It is represented by a number, where low numbers bind more tightly. For example, the numeric expression:

`2 + 3 * 2` is treated as `2 + (3 * 2)`

since `*` has a precedence of 400 and `+` has a precedence of 500.

Associativity is used to resolve ambiguity when there is more than one way to group an expression because there are several operators with the same precedence.
For example,

```
t1 op1 t2 op1 t3
```

would be treated as

```
(t1 op1 t2) op1 t3
```

if `op1` is declared to be *left associative*, or as

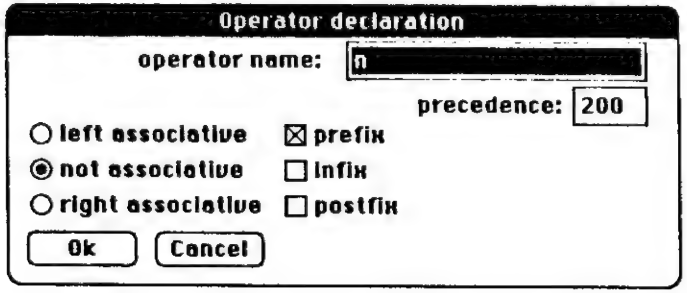
```
t1 op1 (t2 op1 t3)
```

if `op1` is declared to be *right associative.*

If `op1` is declared as non-associative then such an unbracketed expression is treated as a syntax error.

12.1 Declaring operators

Select the **New operator...** command from the **Windows** menu and fill in the dialogue as shown below.



This declares n to be a non-associative, prefix operator with a precedence of 200. Click the **Ok** button.

12.2 Altering an operator declaration

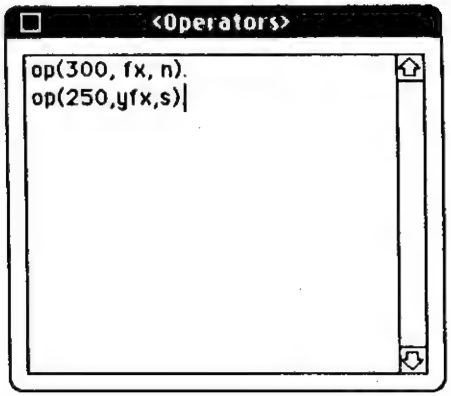
Having entered an operator definition you might want to change it. Once you have declared an operator you will find that an **<Operators>** window has been created. Bring this window to the front using the **Select window...** command. The **<Operators>** window contains each operator that has been declared.

The `fx` in the declaration of n represents the fact that the operator n is prefix and non-associative. Different definitions of an operator are represented by different operator symbols. For example, `xfy` indicates an infix, right associative operator. For a full list of the symbols used you should refer to the Syntax chapter in the MacPROLOG Reference Manual. The declaration for the operator n indicates that it has a precedence of 200.

The details of an operator can be altered by editing the clauses in the **<Operators>** window.

Change the precedence of the operator n to 300 and enter a new operator declaration for s, as shown below.

Select the **Check program** command from the **Eval** menu to recompile the **<Operators>** window.



When you **Save...** a program in source form the **<Operators>** window is saved before any program window. So when you reload the program it will be the first window to be recompiled. This special treatment of the **<Operators>** window ensures that your operator definitions are processed before any use of an operator.

If you save a program as object code, then operator declarations (as in the **<Operators>** window) are *not* saved. To ensure that operators which will be used in input or output terms during the execution of the program are defined you must use MacPROLOG's Execute-on-Load facility and declare the operators inside the definition of the `'<LOAD>'` program. (For further details see the chapter on Implementing an Application in the Reference Manual.)

Changing the precedence of the operator n to 300 will have no effect on the example program. In general, though, changing the declaration of an operator can change the way the MacPROLOG compiler parses the source of a program.

Notice, however, that now n has been declared as an operator, noun terms output from the parsing program will be displayed for example as

```
n girl
```

instead of

```
n(girl).
```

13. Call Graphs and Graphic Windows

A *call graph* for a relation is a tree diagram showing the relation's structure in terms of the other relations used, directly and indirectly, to define it.

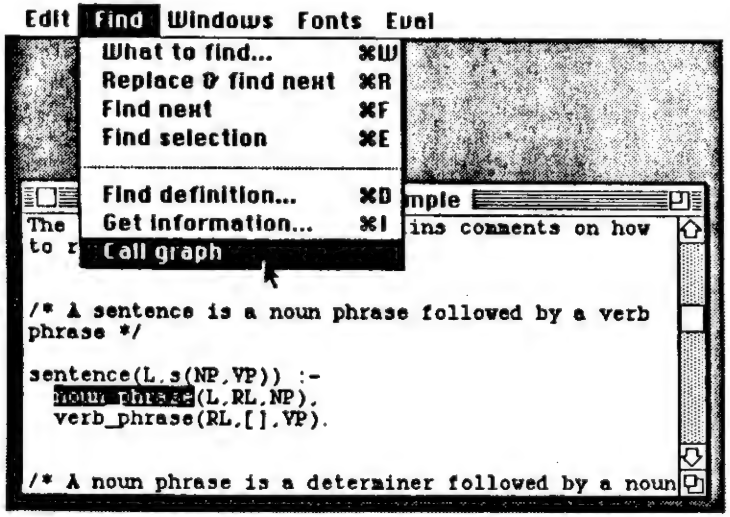
NOTE: Call graphs and graphic windows are only available if you have the MacPROLOG Graphics package. You need to have the file **Graphics Boot** on the same disk as the application file **LPA MacPROLOG™**. (You should *not* try to **Load...** the **Graphics Boot** file. It will be automatically loaded by MacPROLOG when it is needed.)

13.1 Generating a call graph

A call graph may be generated either by selecting the **Call graph** command of the **Find** menu, or by clicking on the **Call graph** button of the **Get information...** dialogue. If you choose the former method then the relation for which the call graph will be generated is selected in the same way as for the **Get information...** or **Find definition...** commands. This means that it will be either the currently highlighted relation name in the front window, or the relation name next to the cursor in the front window, or you will be asked to select a relation name from a scrolling menu of all known relations.

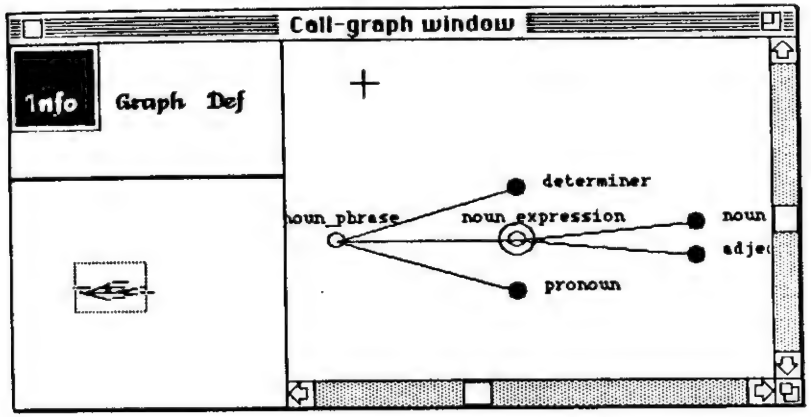
We shall look at the call graphs for some of the relations in the **Guide Example** (the parsing program) mentioned earlier in this guide.

First, highlight `noun_phrase` in the front window (or move the cursor next to it) and select **Call graph** from the **Find** menu.



If it is the first time you have used any graphics this session then the Graphics Boot file will be automatically loaded. You will see a dialogue box telling you that it is being loaded.

The following **Call-graph Window** will appear (it may take a few seconds).



(You can resize the window and use the scroll bars to get this appearance.)

This window is a *Graphic Window*. We shall come back to the structure of a call graph, but first we will look at the various features of a graphic window. This is only a brief introduction to graphic windows - see the Reference Manual for further details.

13.2 Anatomy of a Graphic Window

The graphic window you see is divided into three main areas.

The area to the right, containing the call graph and the scroll bars, is the **viewing pane** of the window. This displays a small part of the whole **drawing area** associated with the window. Notice that the appearance of the cursor changes when it is in the viewing pane.

In the bottom left hand corner of the window is the **viewer**. The viewer is a scaled-down representation of the whole drawing area, and the part currently being displayed in the viewing pane is indicated by the grey rectangle in the viewer.

The left hand side of the window is the **tool pane**, where a window's tools are displayed. The tools are used to manipulate the window's contents in various ways. Each tool is defined by a MacPROLOG program. A tool is activated by clicking on it in the tool pane, and then subsequent clicks in the viewing pane will invoke this tool's program.

The vertical line separating the tool pane from the viewing pane is the **split**.

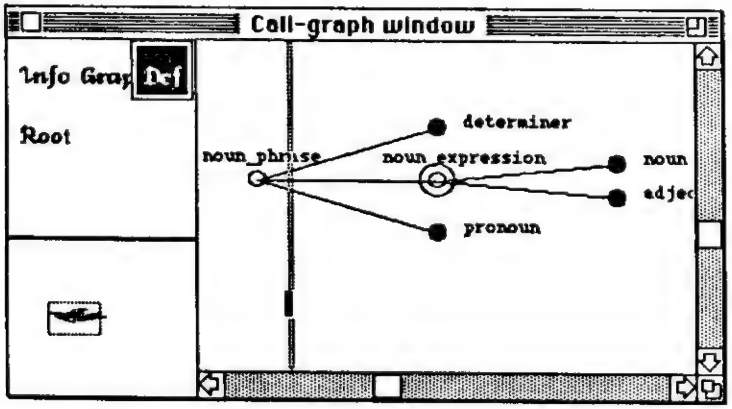
13.3 Manipulating a Graphic Window

You can change the attributes of a graphic window using the mouse.

One way of scrolling the viewing pane is by using the scroll bars. As it is scrolled, different parts of the drawing area are brought into view in the viewing pane. You will also see the grey rectangle in the viewer moving around accordingly.

Alternatively, you can move the grey rectangle in the viewer directly. Click down on the rectangle and drag it. When you release the mouse, the viewing pane of the window will display the part of the drawing area corresponding to the new position of the viewer. (This part of the drawing area may contain nothing at all!) This is a good way to scroll a long way very quickly - the "fine tuning" can then be done with the scroll bars.

The split can also be changed by clicking on it and dragging it to the right or the left. You have to click *exactly* on the split line. The cursor will change to a thick vertical bar and you can then drag a grey image of the split to a new position.



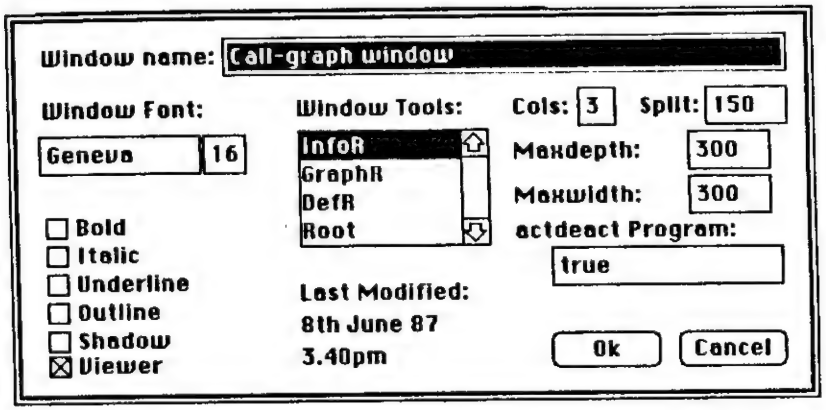
By dragging it to the far left or far right you can make the graphic window either all viewing pane or all tool pane if you wish. In general, tool pictures will shrink or expand when the split is moved, but in this case the tool pictures are text, which does not get scaled in the same way. The picture in the viewing pane will not be scaled, however - just more or less of it will become visible.

The split can also be changed in the **Window details...** dialogue described below.

13.4 Graphic Window Details

With the **Call-graph window** at the front, select **Window details...** from the **Windows** menu. Because it is a graphic window, the dialogue you will see displayed is different from the standard one for program edit windows.

In this dialogue you can examine and alter the various attributes of the window.



The **Window Font** field contains the default font that will be used for any text subsequently added to the window. (You cannot change the text already there in this way - see the Reference Manual for further details).

The **Window Tools** field shows a scrolling list of the tool programs associated with the window. In this graphic window we have the tool pictures Info, Graph, Def and Root, whose corresponding MacPROLOG programs are called `'InfoR'`,`'GraphR'`, `'DefR'`, and `'Root'`.

The **Cols** field determines how many columns will be used to display the tools in the tool pane. If you have a narrow tool pane (if you have moved the split over to the left) it may be better to change this to 1 or 2 columns, so the tools' pictures do not appear squashed.

The **Split** field is the number of pixels from the left of the window that the split line is positioned. Changing this number is equivalent to using the mouse to drag the split in the actual window.

The **Maxdepth** and **Maxwidth** fields indicate the size of the drawing area, and may be changed. (See the Reference Manual for further details).

The **actdeact Program** is an advanced option which we will not cover here - see the Reference Manual for a full description.

The **Viewer** check box at the bottom determines whether or not the viewer is displayed. Try switching it off to see the tool pane expanded to the full depth of the window.

13.5 Call graph tools

The tools in the tool pane of the call graph window are activated by clicking on them. The **Info** tool is probably already active (it is displayed in reverse video) - if it is not, click on it. Now click on any of the nodes of the call graph. This has the same effect as the **Get information...** command of the **Find** menu so you will see the information dialogue for the relation whose node you clicked on.

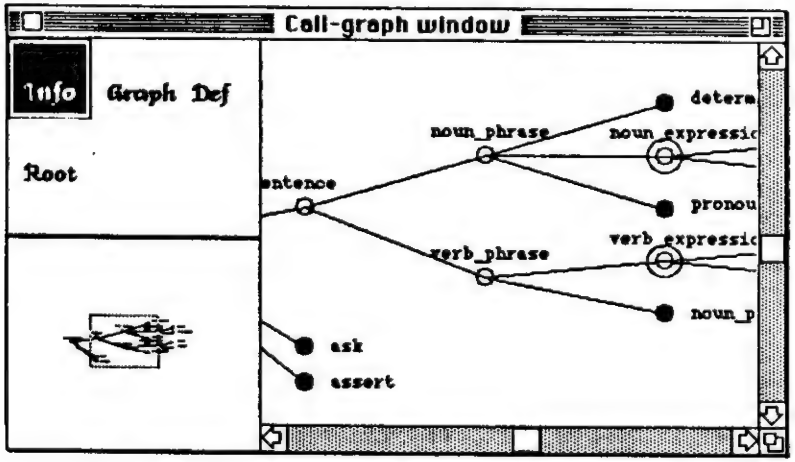
Similarly, if you activate the **Def** tool by clicking on it and then click on one of the nodes in the call graph, this has the same effect as the **Find definition...** command for that relation.

Now scroll the viewing pane so that the left hand side of the call graph is not displayed. Click on the **Root** tool and click anywhere in the viewing pane. (If the **Root** tool is not visible then drag the split or resize the window until it is.) This will bring the "root" node of the call graph back into the viewing pane. This may well be useful when you have a very large call graph displayed. (You can achieve the same effect by dragging the viewer rectangle to an appropriate position.)

Finally the **Graph** tool will display the call graph for any of the existing nodes. Again, this is most useful for very large call graphs, when you may wish to isolate one particular part of a complex graph.

13.6 Call graph nodes

Below is the call graph window for the relation `add-sentence`.



A call graph has several different nodes representing the relations called by the root relation. The nodes have different graphical appearances depending on the type of relation.

A hollow circle represents a normal non-recursive program (such as `verb_phrase`).
A double circle represents a directly recursive program (such as `noun_expression`).
A grey filled circle represents a program defined by assertions (such as `determiner`).
A circle filled with horizontal stripes represents a System primitive (`ask` and `assert` above).
A filled black circle represents a relation which has already appeared in the call graph and is therefore not expanded again (`noun_phrase` above).

14. Printing

You may print the windows of a MacPROLOG program if you have a printer attached to your Macintosh. If you have the MacPROLOG Graphics package you can also print graphic windows, with a choice of printing formats.

First select the **Page setup...** command from the **File** menu to check that the page size and other options are correct for the printer you are using.

The **Print...** command only prints the *visible* windows. There may be a mixture of different window types visible (e.g. program, display and graphic windows), which will all be printed. Remember that a *visible* window is one which is somewhere on the screen, even if it is obscured by another window on top. If a window is being printed, all its contents are printed (and not just the portion of the text which you can actually see). Note that the **Default Output Window** will also be printed if it is visible, so be sure to **Hide** it if you don't want it printed.

When you select the **Print...** option from the **File** menu, you will be presented with a standard Macintosh dialogue from which you can select the print quality, the number of copies to be printed, and so on. (The exact format of this dialogue depends on the printer you are using.)

As the printing is being done, a dialogue box will be displayed giving the name of the window currently being processed. You may press **⌘.** to stop the printing, but the current contents of the print buffer will still be sent to the printer when you do this and so the printing may not stop immediately.

14.1 Printing Programs

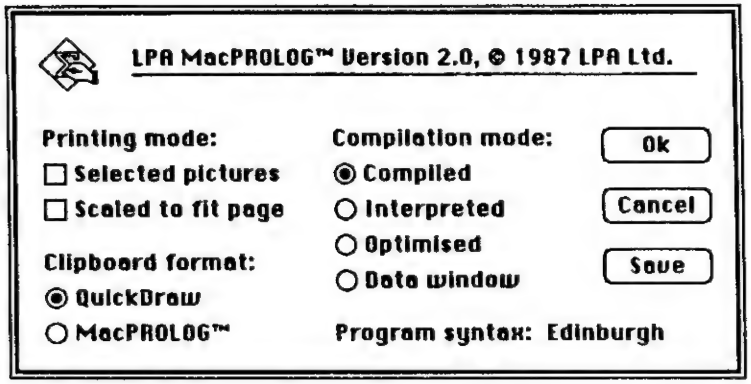
The pages of the text or program edit windows will be numbered consecutively, with the window name, program name and date printed across the top of each page. Each window will be started on a new page, in the same font, style and text size which appears in the window on the screen. If a window requires more than one page there is supplementary page numbering within that window's pages. Graphic windows are not included in the page numbering.

14.2 Printing Graphics

Only the contents of the drawing area of a graphic window is printed. There are four possible formats for this printing, which you may set in the **Defaults...** option of the **File** menu.

(In the **Page setup...** dialogue there may be additional options for scaling or rotating the final printed image, depending on your printer. This is standard for most Macintosh applications.)

The **Defaults...** dialogue contains two check boxes, **Selected pictures** and **Scaled to fit page**, each of which may be on or off. This represents four possible graphics formats as follows.



1.

☐ Selected pictures
☐ Scaled to fit page

The whole of the drawing area of a graphics window will be printed without scaling, spread over multiple pages if necessary. Pictures themselves may well become split over more than one page.

2.

☐ Selected pictures
☒ Scaled to fit page

The whole of the drawing area associated with a graphics window will be scaled to fit onto a specified number of sheets of paper. The default number of pages is one, but you can change this (see below). All pictures are shrunk accordingly.

3.

☒ Selected pictures
☐ Scaled to fit page

Only the currently selected pictures in a graphics window will be printed. They will appear in their 'true' size on one sheet of paper, with the first picture being in the top left hand corner of the paper. **NOTE :** If the picture/s are larger than the sheet of paper, you will lose some of the image.

4.

☒ Selected pictures
☒ Scaled to fit page

Only the currently selected pictures in a graphics window will be printed. They will be scaled to fit on a specified number of sheets of paper. This may involve expanding or shrinking the pictures depending on how large an area the selected pictures cover.

When the pictures are scaled to fit the page (options 2 and 4 above), the default number of pages on which to print is one. To change this, so that more than one page is used, you need to change a MacPROLOG environment property using a `set_prop` call.

The following call will set the number of pages to *number*.

```
set_prop('DEFAULT','PAGES',number)
```

The picture will be printed as "long and thin", i.e. its width will be scaled to fit the width of one sheet of paper, and its height will be scaled to fit the length of *number* sheets of paper.

14.3 Scaling Text

Within a graphic window, text described by a MacPROLOG text descriptor such as `text` or `textbox` cannot be scaled in printing, even though the surrounding pictures might be. However, if you **Cut** a picture from a graphic window and then **Paste** it back in, using the **QuickDraw** format set in the **Defaults** dialogue, this will convert all text in the picture so that it can be scaled on printing. Note, however, that this action loses the MacPROLOG term description of the picture.

A call graph is automatically created in QuickDraw format, so its text will be scaled if necessary on printing.

15. The Optimising Compiler

If you have purchased the optional Optimising Compiler you can use it instead of the built-in compiler to compile the programs in a window. The optimising compiler is itself a compiled MacPROLOG program. It generates code which executes faster than code generated by the built-in compiler. The file **Optimizer Boot** must be on your LPA MacPROLOG™ disk.

In our parsing program example the dictionary lookup is used to check that a given word is of a particular category (`noun`, `verb`, etc.). This is one area where the optimising compiler can do some optimisations by generating indexed code. The indexing scheme uses the values of the first argument in each clause of the program, and will exclude those clauses which cannot possibly match a run-time call. This narrowing down to a subset of clauses can significantly speed up the search for a matching clause.

First the code generated by the Optimising Compiler switches on the type of the argument, distinguishing between number, constant, empty list, non-empty list and compound term types. So, when a call has an empty list as a first argument, only those clauses with an empty list or variable in the first argument position in the head of the clause will be tried.

A second switch occurs if there are more than two clauses with a constant as their first argument, or more than two clauses with an integer as their first argument. For these clauses there is a further switch on the value of this first argument. So, in the case of our dictionary lookup, the attempt to show that `noun(boy)` is true will result in a switch to the one and only matching clause.

Note, however, that this indexed code is *not* produced if the relation has more than two clauses which contain a variable as their first argument.

The optimiser also generates faster code for the unification of arguments. This, coupled with other optimisations, results in faster execution of recursive programs. As an example, the famous naive reverse bench mark, which benefits both from indexing and the faster unification code, runs three times as fast when optimised.

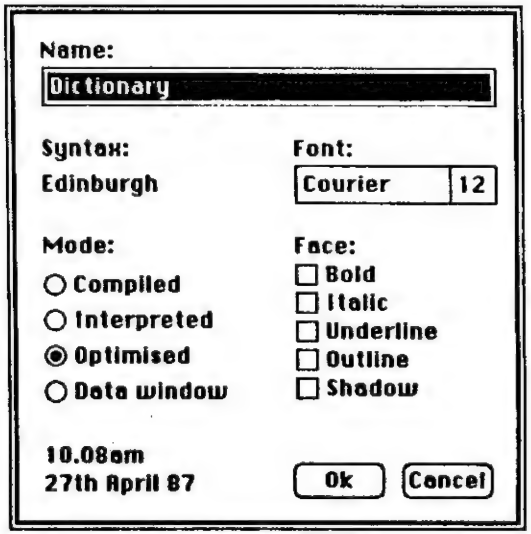
The Optimising Compiler takes longer to compile a program than the standard built-in MacPROLOG compiler. For this reason you will probably only want to compile to optimised code in the final stages of development of a program, using the built-in compiler for quicker development in the earlier stages.

15.1 Using the Optimising Compiler

To optimise the code of a window, change its compilation mode in **Window details...** to **Optimised** and then recompile the window. Whenever an **Optimised** window is to be compiled, the Optimising Compiler will be called.

WARNING: The first time the Optimising Compiler is needed in a MacPROLOG session it will be automatically loaded from disk. You should **NOT** try to load it yourself using the **Load...** command.

Bring the **Dictionary** window to the front and change its compilation mode to **Optimised**. Recompile the contents of the window using the **Check program** command.



16. Porting Programs

16.1 Importing Edinburgh syntax programs

When you first start to use MacPROLOG you may have some existing PROLOG programs which you want to use. These may have been transferred as ASCII text from some other computer, or developed and saved as a MacWrite text file. There are two ways of loading these programs into MacPROLOG.

1. Using the **Load...** command

When you use the **Load...** command the names of the ASCII text files on your disk are displayed along with the normal MacPROLOG source and object code programs. If you click on one of the text files MacPROLOG will automatically detect that it is an ASCII file and will try to load the text into a single edit window. If the text file is longer than about 30K, more than one window will be created. You will see a dialogue asking if you wish to continue loading the file into more than one window. The text is not processed in any way as it is read in and it is therefore possible that the text may not be split conveniently across windows. You can remedy this after it has been loaded using **Cut** and **Paste**.

The name of the window(s) will be the same as the name of the file. You may then wish to use the text editing facilities of MacPROLOG to rearrange your program text into several program windows before saving it as MacPROLOG source. Remember that before you can compile your program, clauses which share the same relation name must be contiguous within one program window.

There is a MacPROLOG program **Importer Utility** in the **Utilities Folder** on the MacPROLOG disk that will read in a text file and automatically arrange relations contiguously in windows. You may find this useful if you have a very large text file in which the different clauses for a relation are mixed with clauses for other relations. The utility is a source program which you can modify for your own purposes. A comments window explains how to use the program.

2. Using the `consult` primitive

Text files can also be loaded using `consult`. To use this primitive the file must contain Edinburgh syntax clauses, commands, or definite clause grammar rules. `consult` does not generate an edit window. The clauses are stored as dynamic code (as if they had been added using `assert`), and the commands are executed as they are encountered. Note that in this case any comments contained in the file will be lost as the file is read in.

If you have used `consult` you can if you wish create an edit window yourself and then use the `listing` primitive to get the source text into this window. The program can then be saved as MacPROLOG source. For example, if you use **New...** to create a window called **mywindow**, then `consult` your file, you can list the relations in this window as follows.

```
tell(mywindow),
listing,
told.
```

If you have several relations defined it is probably better to create more than one program window and put a few relations in each window. (The call to `listing` will generate a scrolling menu of relation names from which you can select the ones to be listed.)

16.2 Using MacPROLOG™ Version 1.0 Source Programs

There are some differences between MacPROLOG™ version 1.0 and version 2.0 - the names and arities of some of the system primitives have been changed. However, to facilitate the use of 1.0 programs in version 2.0, there is a file called **1.0 Compatibility** in the **Utilities folder**. If you load this file you will find the instructions for its use and details of the changes in a comments window.

MacPROLOG version 2.0 will load version 1.0 source files, both in Edinburgh and LPA Standard syntax. However, you cannot load version 1.0 object code files. Further information on the use of LPA Standard syntax in version 2.0 is given in the comments window of the **1.0 Compatibility** program. Because there have been minor changes to MacPROLOG's internal representation of windows, your version 1.0 windows may load into version 2.0 with a different text style. You can change this after they have been loaded, either from the **Fonts** menu or the **Window details...** dialogue.

You cannot load version 1.0 Simple syntax source programs into version 2.0. To port these programs, you should use version 1.0 to convert the source into LPA Standard syntax or Edinburgh syntax and then load this new source into version 2.0.

16.3 Exporting Programs

To transfer an LPA MacPROLOG™ program to another PROLOG system you will have to save the source of your program in a text file. Use the **Text only** option of the **Save...** dialogue, which enables you to save the text of selected windows. You can also save the contents of the **<Operators>** window in a separate text file but remember to convert the `op` clauses of this window into commands.

LPA MacPROLOG™
Environment Guide

Part B : Menu Reference

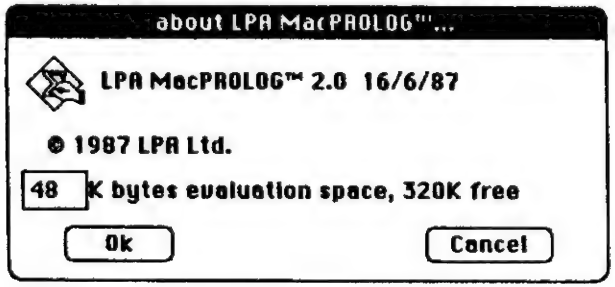
This part of the Environment Guide gives a summary of the menus available within MacPROLOG.

The menu

The top item of the menu tells you about this version of LPA MacPROLOG™.

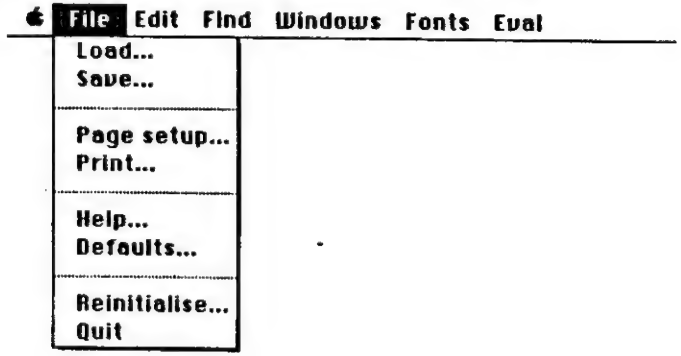
The rest of the menu contains the Macintosh desk accessories of your system disk and will vary according to which ones you happen to have installed. They will include the **Scrapbook** which can be used for transferring information between other applications and LPA MacPROLOG™. You may transfer text, and if you have the MacPROLOG Graphics package you can also transfer pictures.

Selecting the **About LPA MacPROLOG™...** item will display a window telling you what the current version is, how much space is allocated to running programs and how much other memory is available.



You can change the amount of space used for evaluating queries and compiling windows by altering the number shown in the **K bytes** box. The latter figure, in this case 320K, gives the estimated free memory available for edit windows and for the storing of compiled and interpreted code. Any attempt to set an evaluation space of less than 24K will be ignored, as will any attempt to set it to more than the amount of space shown as being free.

The **File** menu



Load...

The **Load...** command is used to load a MacPROLOG file. You will see a dialogue with a scrolling menu containing the names of all the MacPROLOG files and all text files on the current volume. The current volume is either an entire disk, or a folder on a disk (if you have the Hierarchical File System).

The dialogue enables you to switch drives and change disks. With recent versions of the Macintosh Finder you can also change folders on a given disk . Once you have chosen a file from the menu it will be loaded. The type of the file you have chosen is automatically detected and the file is loaded in an appropriate way.

Text files less than 32K are loaded into a single edit window. The name of the window is the name of the file. The text file may have been generated in MacPROLOG using the **Save...** command (with the **Text only** option), or it may have been transferred to MacPROLOG from another application. ASCII text files will not be automatically compiled on loading. This allows you to manipulate the text in the window, or even to split it over several windows, before compiling it with **Check Program**.

If a text file is longer than 32K, you will be asked as it is loading if you want it split into two or more windows. The split will not necessarily be at a term or word boundary, but you can use the editing facilities of MacPROLOG to rearrange the text if necessary after loading.

Both source and object code MacPROLOG program files must have been generated using the **Save...** command. They will be loaded in the same state in which they were saved.

Save...

The **Save...** command can be used to save the source of your program (the edit windows and their textual contents), to save the object code of your program, or to save merely the text of your program.

A dialogue is displayed enabling you to select a file name and the save format. The dialogue will show the name of the last file that you loaded or saved.

The three radio buttons **Source**, **Object code** and **Text only** behave as follows.

If you select **Source**, then your program will be saved as a collection of edit windows in a single file. This means that when you load it again, your program windows will reappear on the screen in exactly the same state (i.e. size, position, textual contents) in which they were saved. The windows will be compiled as they are loaded. You *cannot* edit source files using an external editor. You can only edit them inside MacPROLOG.

If you select **Object code** then only the compiled code of your program will be saved, in a special binary format. The static code (for programs in Compiled and Optimised windows) *and* the dynamic code (for programs in Interpreted and Data windows, and all asserted clauses) will all be saved in the single file. When you load it again you will be able to make calls to the programs but you will not be able to edit the source in a window. The reloaded object code for interpreted relations will still be dynamic code: you will still be able to `assert` to or `retract` from these programs. A code file's icon as displayed by the Finder will have the name CODE on it.

If you select the **Text only** option then just the text of your program will be saved. No code or window details are saved with it. This means that you can then load the file into another application (MacWrite for example). You can save the file as text from this different application, and load it back into MacPROLOG.

If you click on the **Save** button then all your windows, code or text will be saved. This includes invisible windows.

If you click on the **Selective save** button then what happens depends on which of the above radio buttons you have selected.

Source - only the *visible* windows will be saved to the file.

Object code - a scrolling menu will appear from which you choose the relations you want to be saved to the file. The code for these selected relations will be saved as will the code for all the relations directly or indirectly called by these relations. This means that when you load the file again all the necessary code will be there.

Text only - a scrolling menu will appear enabling you to choose one or more windows. The text of these selected windows will then be saved in the file in the order of the names in this menu.

If you are re-saving a file, you may like to keep a backup of the previous version of the file. The **Make backup** option box of the **Save...** dialogue allows you to specify whether or not a backup of the original file should be made. If this option is selected then the original file will be copied to a file called **Backup of ...** before the new copy is saved. If the **Make backup** option is not selected the original file is erased.

If a file of the name you have chosen already exists on the disk you will still be asked if you want to replace that existing file whether or not you have chosen to make a backup.

Page setup...

This command displays a standard Macintosh dialogue enabling you to specify the page layout for printing. The exact form of the dialogue will depend on the printer which you have installed. For further details you should refer to the Macintosh manual or your printer manual.

Print...

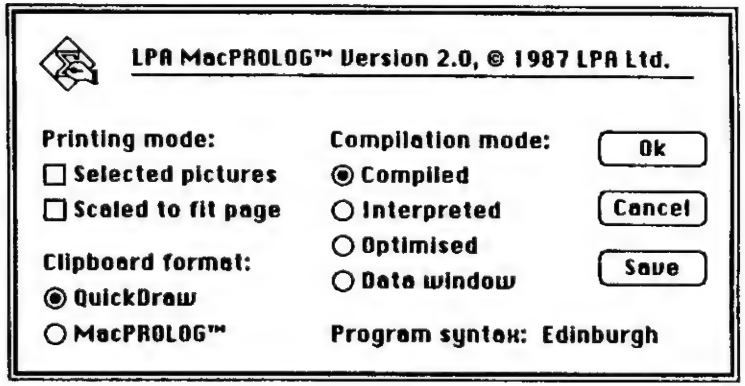
This command will print all the currently visible windows (including the **Default Output Window** if it is visible). A standard Macintosh dialogue is used to specify print quality, type of paper, number of copies etc. The exact form of the dialogue will depend on the printer that you have installed.

Help...

A dialogue is displayed with a **Help Mode** button. If you click on this button then you enter Help Mode, and whenever an item is selected from one of the **Edit**, **Find**, **Windows** or **Eval** pull-down menus a short explanation of that item is given. To switch off Help Mode and return the menus to their normal behaviour simply select the **Help...** menu item again.

Defaults...

A dialogue is displayed allowing certain global characteristics to be viewed and changed. If you click on the **Save** button the new characteristics are saved to the LPA MacPROLOG™ application file (and will be reloaded next time you use MacPROLOG), otherwise they only apply for the duration of the current session.



The **Clipboard format** and **Printing mode** are only applicable if you are using the MacPROLOG Graphics package. The **Clipboard format** determines whether pictures in the clipboard are represented graphically as QuickDraw pictures, or textually as MacPROLOG picture descriptions.

The **Printing mode** determines how a graphics window should be printed (this is explained in more detail in "LPA MacPROLOG™ : Getting Started").

If you change the **Compilation mode** for windows, then any new windows you subsequently create will be in this new mode.

A **Compiled** window will have all the clauses for a given relation compiled into a contiguous block of object code. You cannot update this object code using the data base primitives such as `assert` and `retract`, nor can you retrieve the source of a particular clause using `clause`. The only way you can change the code is by editing the window and recompiling it.

An **Optimised** window is similar to a compiled window except that it is compiled using the Optimising compiler. You must have the Optimizer Boot file to use this option.

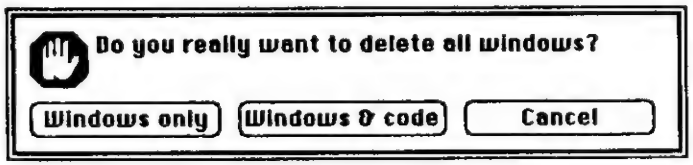
An **Interpreted** window will have its relation definitions compiled a clause at a time, but with all the clauses for a relation linked. In consequence, the definitions can be accessed and updated using the various data base primitives such as `clause`, `assert` and `retract`.

A **Data** window is the same as an interpreted window except that when its clauses are altered in any way using the MacPROLOG data base primitives, the text of the window is altered to reflect these changes. A **Data** window always shows the current state of the data base relations defined within it.

The relations defined in Compiled and Optimised windows are *static*. Those defined in Interpreted and Data windows, and those defined solely by asserted clauses, are *dynamic*.

Reinitialise...

This command reinitialises the system by killing all the program windows. You may also choose to get rid of all your existing compiled and interpreted code. You will see a dialogue with three buttons including a **Cancel** button.



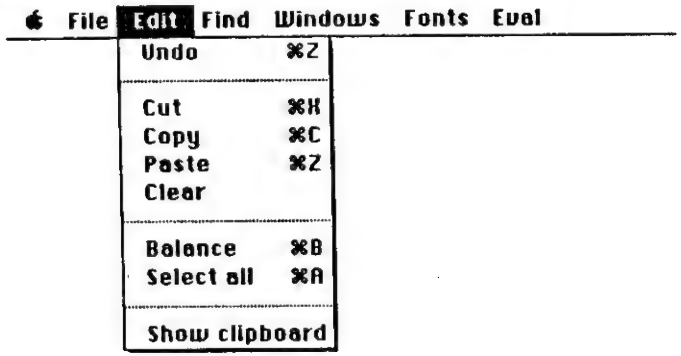
If you select the **Windows only** button then all your windows will be killed but any compiled or interpreted code that has been created by compiling the window will still be present, so you may make calls to these relations although you cannot edit their source.

If you select the **Windows & code** button then all your windows will be lost and all your programs will be cleared from memory.

Quit

The **Quit** command exits MacPROLOG and returns you to the Finder. If you have edited any windows since the last **Save...** command you will be asked if you want to save your program first.

The **Edit** menu



The **Edit** menu controls the main editing commands. The first group of commands implements the standard Macintosh "Cut 'n' Paste" edit model, the next group controls the selection of text, and the last item allows the clipboard window to be displayed on the screen.

Undo

The effect of this command is to undo your last editing action in the front window, and return the window to its previous state. For example if you have just cut a paragraph of text by mistake then selecting the **Undo** command will put back the text you have just cut.

Note that he clipboard is not affected by an **Undo** operation (so, for example, you cannot undo a **Copy** operation).

Cut

The **Cut** command removes the currently selected text (or selected pictures) from the front window. **Cut** has no effect if nothing is currently selected. **Cut** can also be used in a desk accessory or dialogue. The clipboard will contain the last item cut or copied.

Copy

The **Copy** command is similar to the **Cut** command except that the text is not deleted. It is copied to the clipboard. When pictures are copied, the contents of the clipboard will depend upon the default picture setting (see **Defaults...**).

Paste

The **Paste** command inserts the current contents of the clipboard into the front window. If the front window has text selected then the contents of the clipboard will replace this selected text. Otherwise it is inserted after the cursor. Similarly, when pictures are pasted into a graphics window any existing selections are first deleted.

Clear

This deletes the currently selected text/pictures in the front window, *without* putting it in the clipboard. Used in conjunction with **Select all**, it can be used to clear an entire window or edit field. It is useful to periodically clear the **Default Output Window** to free the memory that is being used to store this window's contents.

Balance

This extends the currently selected text to include the next outer pair of matching brackets. The bracket characters recognised are ( ) [ ] { }. **Balance** does not look for an exact bracket match. It will match an opening (with a corresponding closing]. It also counts brackets inside quoted constants, so it will not be accurate for text in which you have unmatched brackets in quoted constants. If the cursor is immediately to the right of a closing bracket or to the left of an opening bracket it will match to the corresponding opening or closing bracket. You can cancel the highlighting by clicking on the mouse anywhere in the current window or edit field.

Select all

The **Select all** command selects all the text/pictures in the current window or dialogue edit field.

Show clipboard

The **Show clipboard** command displays the (currently hidden) clipboard window. This window always reflects the current contents of the clipboard.

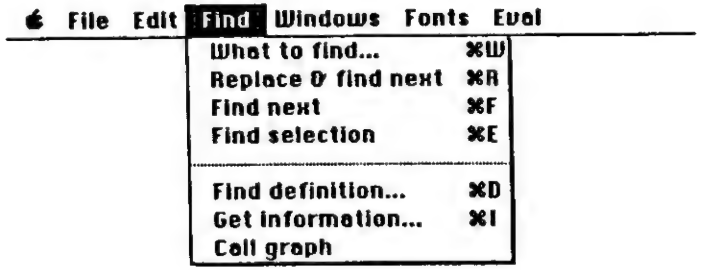
You may notice that if you show the clipboard at the start of a session, it will contain the clipboard contents from a previous application. This is one way of transferring text between another application and MacPROLOG.

(You can also cut and paste to and from the **Scrapbook** in the menu.)

Hide clipboard

This item replaces the **Show clipboard** item in the **Edit** menu whenever the clipboard is visible. The clipboard can be closed by clicking the 'go away' box on the window, or by selecting **Hide clipboard** from the edit menu.

The **Find** menu

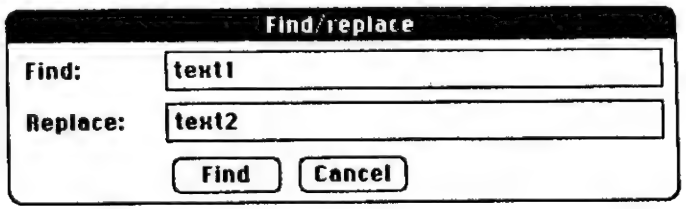


The **Find** menu includes search and replace commands for text. When text is being searched for, all program edit windows are searched (including invisible ones). When the last window has been scanned the search continues at the first window again, and so on.

In addition, there are three commands for getting information about MacPROLOG programs. These last three commands **Find definition...**, **Get information...** and **Call graph** all use a common procedure to determine the name of the program to process. First, the current selection range of the top window (if any) is examined. If this is not a relation name then the atom (if any) nearest to the current cursor is considered. Finally, if neither of these yield the name of a relation, a scrolling menu is displayed containing the names of all known relations, and you are asked to select one of them. The **Find:** field of the **What to find...** dialogue is also updated to show the relation name determined by this procedure.

What to find...

The **What to find...** command displays a dialogue in which you specify the text to be found in the **Find:** field and the text with which to replace it in the **Replace:** field. The **Replace:** field may be left blank. The fields will show the last text you used to find and replace.



Replace & find next

The currently selected text in the front window is replaced by the the contents of the **Replace:** field. The selected text is often text found by a previous search, but it need not be. It could, for example, be a cursor insertion point or some other selected text.

After the text has been replaced the next occurrence of the **Find:** text is searched for. All windows are searched in turn, in a circular fashion, even if they are hidden.

Find next

The **Find next** command simply searches all the program edit windows for the next occurrence of the text in the **Find:** field, and highlights it when it is found.

Find selection

This will find the next occurrence of the selected text of the current window. The text may have been selected by dragging the mouse across the text, or as a result of some other command. The search starts from the end of the selected text within the current window but will go on to other windows if the text is not found.

Find definition...

The **Find definition...** command brings to the front the window in which a relation is defined and then repositions the cursor at the first occurrence of the relation name in that window. If this is not the head of the first clause, a **Find next** (or ⌘F) can be used to search forward for the first clause.

The relation name is either determined by the current text cursor position or is chosen from a menu of known relation names. (See above for further details of this procedure).

Get information...

This command displays a dialogue window showing various characteristics of a relation. This includes the name of the window in which it is defined (if any), the mode of compilation (one of Compiled, Optimised, Interpreted or Data), the relations's arity or arities (if known) and a scrolling menu of all the other relations which occur in its definition. The dialogue also allows you to set or unset a spy point on the relation, to see its associated call graph, or get information on the relations which it calls.

There is also an edit field in which you may enter text (usually your own comments about the relation). This comment is stored as a clause for the 'COMMENT' relation of the MacPROLOG environment, and also in the **<Comments>** window. It is redisplayed when you next select **Get information...** for this relation. You can change the comment by editing the text either in the **Get information...** dialogue or in the **<Comments>** window.

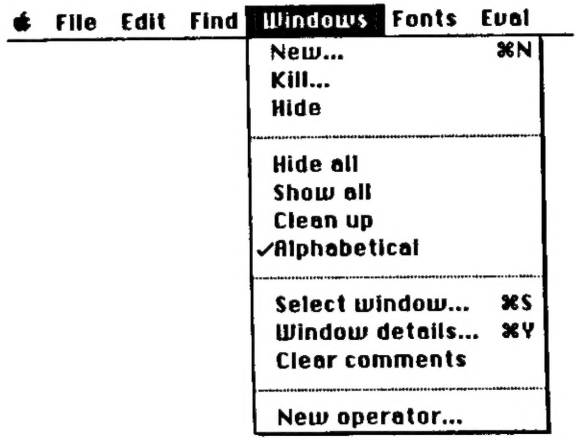
The relation for which the above information is displayed is determined in the same way as for **Find definition...**

Call graph

This command generates a graphic window portraying the hierarchical structure of a program. Tools in the graphic window enable you to get information or find the definition of relations with nodes appearing in the graph. The relation at the root of the call graph is determined by the procedure described above.

This will only work if you have the Graphics Boot file on the MacPROLOG start up disk. If the Graphics Boot file is not already loaded then it will automatically be loaded when you select this menu item.

The **Windows** menu



The **Windows** menu includes commands that control your program's windows. You may create, select and delete windows, and also declare operators and change the characteristics of windows.

New...

The **New...** command is used to create a new program edit window. A dialogue will be displayed asking you to enter the name of the window.

The name must be different from the name of any other current window. The compilation mode (**Compiled, Interpreted, Data** or **Optimised**) will be as recorded in the current **Defaults...** dialogue.

Kill...

The **Kill...** command is used to delete the front window. For program windows you will get a dialogue with command buttons **Window only, Window & code,** and **Cancel**. If you select the **Window only** button then only the window and its source text will be deleted. The associated compiled or interpreted code for the window will remain. To remove this as well you need to select the **Window & code** button.

Hide

The **Hide** command will make the front window invisible. It does not delete the window or its text, or any of its associated compiled code. The same effect can be obtained by clicking the *go away* box of the window. The **Hide** command will act on any window, including the **Default Output Window** and the **Clipboard** window.

Hide all

All visible windows (including all program and graphic windows, the **Default Output Window** and the **Clipboard** window) are made invisible.

Show all

All invisible windows are made visible. It redisplays all the hidden program and graphic windows, and also the **Default Output Window** if it was hidden.

Clean up

This command neatly stacks the visible windows on the screen so that the title bars of each of them can be seen. The process is similar to stacking a sheaf of papers. Windows are stacked in groups according to the window type.

Alphabetical

Certain commands (**Show all**, **Clean up** and **Select window...**) operate on groups of windows. Each group corresponds to a different window type (*program*, *information*, *display*, *graphic* and *dialogue*).

When the **Alphabetical** item is ticked, the windows within each group are alphabetically ordered by their names. When it is not ticked the groups are ordered according to the creation date of each window.

Select window...

As an alternative to selecting a window by clicking on it, the **Select window...** command allows you to bring to the front any of the current windows, whether hidden or visible.

You may select one or more windows from the scrolling menu of all known windows, and each is brought out to the front. Of course if more than one window is selected only one (the last) will end up as the front window.

The **Select window...** command is most useful for finding windows which are completely obscured by other windows or which have been hidden with the **Hide** command.

Window details...

This command allows you to view and change the various characteristics associated with the front edit window. It has no effect if the front window is not an edit window. So select the edit window whose properties you want to look at either with the mouse or using **Select window...** before using this command.

You can change the name of the window by editing the **Name:** field of the dialogue.

The **Syntax** field shows the syntax of the window; this cannot be changed.

You can change the font and/or text size of the window by editing the **Font:** fields of the dialogue. If the font name is not recognised then the default system font is used (which is usually **Chicago**). The font size can be any positive integer up to a maximum of 72.

You can change the text face of the window by clicking on one or more of the check boxes labelled **Bold**, **Italic**, **Underline**, **Outline** and **Shadow**.

The compilation mode can also be changed. The options are as explained for the **Defaults...** command.

If you change the compilation mode you should recompile the window using the **Check program** command.

Finally the creation date of the window is displayed.

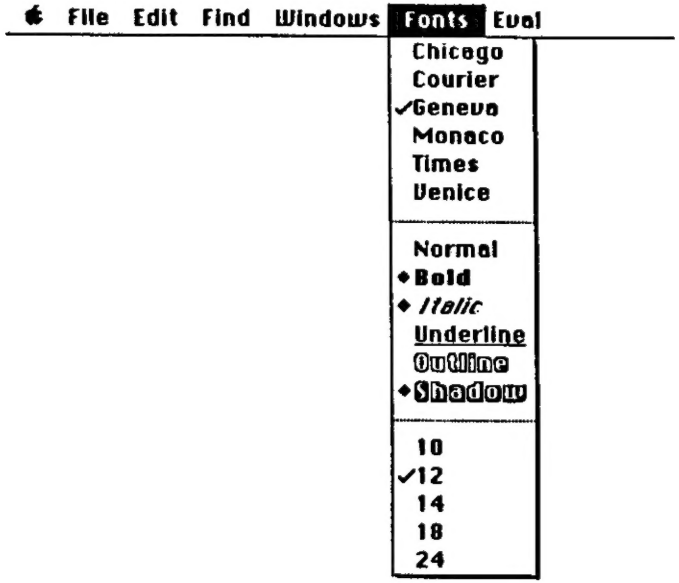
Clear comments

The **Clear comments** command allows you to delete the **<Comments>** information window and optionally its associated database of `'COMMENT'` clauses.

New operator...

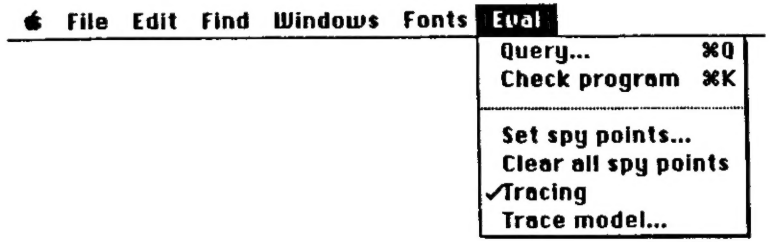
The **New operator...** command allows you to specify a new operator. A special operator declaration dialogue is displayed in which you give the relevant details of the operator: its name, its precedence level, whether it associates to the left, right or not at all, and whether it is infix, prefix, or postfix .

The **Fonts** menu



The **Fonts** menu shows the font details of the front window by marking various items. The font's name is indicated by a tick, its style is indicated by one or more diamonds, and its size is indicated by a tick. As different windows come to the front so the markings in the **Fonts** menu will change appropriately. By selecting an item in the **Fonts** menu you can change either the font, the style or the size of the text in the front window.

The **Eval** menu



The **Eval** menu is used to run queries and to recompile all edited programs. You can set and clear spypoints and also tailor the trace facility to your own individual needs.

Query...

If a **Query...** dialogue already exists then it will be made visible and brought to the front. Otherwise this command creates a new dialogue containing two edit fields labelled **Q:** (query) and **A:** (answer).

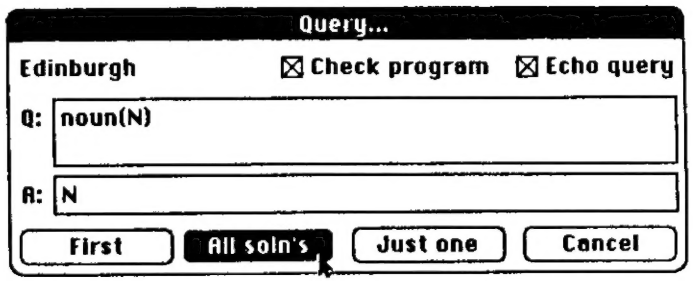
If it is the first **Query...** in a session then the **Q:** field will contain the word **true** and the **A:** field will be empty. Otherwise the fields contain the same text as the previous **Query...** dialogue.

You may fill the **A:** field with any valid term, representing the form of answer you would like. For example, it could be some of the variables of the query, separated by commas. If the **A:** field is left blank, the bindings for all the variables in the query are displayed as the default answer.

The **Q:** field must be a conjunction of conditions of the form

$$atom_1, atom_2, \ldots, atom_k$$

for some $k \geq 1$. A terminating full stop is optional.



The **Check program** check box determines whether or not recompilation takes place prior to evaluating the query.

If the **Echo Query** check box is selected then output will be echoed to the **Default Output Window**, otherwise it only appears in the **Query...** dialogue itself.

You may choose to have **Just one** solution or **All soln's**. Alternatively you can step through the solutions by clicking the **First** button. (The dialogue then changes so you can see the **Next** solution or **The rest**).

Check program

Check program will recompile all program windows which have changed since the last compilation, and all the new windows. Only those windows which have had their text changed or their compilation mode altered are recompiled; others are ignored. The recompilation generates code for a window in accordance with its mode (**Compiled, Interpreted, Data** or **Optimised**). See the description of **Defaults...**

NOTE: Until you use the **Check program** command, (or run a query with the **Check program** option selected) the MacPROLOG™ environment will not be aware of any new relations you have defined. Certain routines such as **Set spy points...** and **Find definition...** may generate scrolling menus which appear to be incomplete if you have not done a **Check program** to compile the windows in which your new relations are defined.

Set spy points...

This displays a scrolling menu of all the relation names for which there is code. Any relations for which a spy point already exists are highlighted. You can change the selected relations by mouse clicking and shift clicking.

The check box labelled **Tracing** indicates whether or not the trace facility is enabled, and behaves exactly as the **Tracing** item in this menu.

Clear all spy points

Spy points are removed from every relation.

Tracing

This item indicates whether or not the trace facility is enabled. When there is no tick against the item, an evaluation behaves as if spy points had been cleared. When ticked, certain information about each call to a spied relation will be displayed in the **Default Output Window**. The information that is displayed depends on the settings of the **Trace model...** (see below).

In addition, for a dynamic relation you are given the option to trace fully the use of its clauses in attempting to solve the call. This gives you more information about the way that your program is running.

Trace model...

This command allows you to choose which of the **Call, Exit, Redo** and **Fail** ports you want displayed during the trace of an evaluation.